An Analysis of Stream processing Languages

Miran Dylan
Department of Computing
Macquarie University
Sydney - Australia
2009

miran.dylan@students.mq.edu.au

5/24/2009

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.

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.

Table of Contents

Glossary

Term	Description
DBMS	A database management system (DBMS), sometimes just called a database manager, is a program that lets one or more computer users create and access data in a database
tuple	a tuple is a sequence (or ordered list) of finite length
DSMS	Data Stream Management System DSMS also called SDMS Stream Data Management System, is a system to deal with high volume of load of unbound data streams where the data is provided on real-time and continuous basis
SQL	SQL (Structured Query Language) is a database computer language designed for the retrieval and management of data in relational database management systems (RDBMS)
STREAM	Stand for STanfordstREamdatAManager, is a prototype DSMS developed by Stanford University.
QoS	Quality of service is the ability to provide different priority to different applications, users, or data flows, or to guarantee a certain level of performance to a data flow.
ESP	Event Stream Processing is a set of technologies designed to assist the construction of event-driven information systems. ESP technologies include event visualization, event databases, event-driven middleware, and event processing languages
CEP	Complex Event Processing is primarily an event processing concept that deals with the task of processing multiple events with the goal of identifying the meaningful events within the event cloud
Aurora	Is a general purpose Data Stream management System developed by Brandies and Brown Universities and The MIT.
Borealis	Is second generation Aurora which implements the multiple processing nodes on a distributed environment. Borealis inherits the core infrastructure from the Medusa project.
Medusa	Medusa is a distributed stream-processing system built using Aurora. Medusa takes Aurora queries and distributes them across multiple nodes.
SPADE	SPADE is a language specification and compiler infrastructure for stream processing applications
CQL	Continuous Query Language is an extended SQL language used in STREAM system.
SQuAl	Stream Query Algebra: is a extended SQL language used in Aurora System
MAIDS	The MAIDS (Mining Alarming Incidents in Data Streams) project is aimed to perform a systematic investigation of stream data mining principles and algorithms, develop effective, efficient, and scalable methods for mining the dynamics of data streams, and implement a system prototype for online multi-dimensional stream data mining applications
SPC	The Stream Processing Core (SPC) provides the scalable, distributed runtime streaming services for System S including efficient stream transport, automatic and application-assisted flow management, failure resiliency, system monitoring and analysis.
CACQ	Continuously Adaptive Continuous Queries (CACQ) a stream processing language designed specifically for TelegraphCQ.
TelegraphCQ	TelegraphCQ is an implementation of the Telegraph dataflow engine.
System S	System S is an exploratory, grand challenge, prototype system being developed by IBM Research to support highly dynamic applications that extract information and knowledge by analyzing enormous volumes of relatively unimportant data
SPADE	SPADE stands for Stream Processing Application Declarative Engine. It is a programming language and a compilation infrastructure, specifically built for streaming systems
COUGAR	Is widespread distribution networked database, it deals with small scale sensors, actuators, and embedded processors, it transform them from physical world into a computing platform.

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 need to address the common desired characteristics for these technologies.

Today, the challenge of dealing with high volume of real-time data requires organizations to think of new ways and to improve their ability to acquire, analyze, and respond to massive amounts of complex event and process data in real-time. New real-time oriented systems used for critical implementations such as military and stock market applications must be able to quickly sense and respond to patterns of interest that indicate imminent or potential threats by constantly sifting through an enormous volume of data and information coming from electronic communications, emails , financial transactions, sensor data and other online sources. Such applications require the capability to capture, correlate, aggregate, filter, analyze and instantly react to events on real-time bases.

A data stream is a real-time, continuous, sequence of items ordered by arrival time or by time stamp. It is impossible to control the order in which the data arrives, nor is it feasible to locally store a stream in its entirety. In addition, queries over streams run continuously over a period of time and incrementally return new results as new data arrives.

Many papers have been published detailing the issues and requirements for such technology. Several prototype and commercial implementations have been produced by different groups (Golab & Özsu, 2003) that have addressed issues in data stream management; 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 report describes the common characteristics and desired features of Data Stream Management System (DSMS) and the underlying language and technology issues as well as highlighting the recent progress on some of the commercial and prototype data stream management systems. It provides a comparison between few of those selected data stream management systems based on the identified characteristics and features and the use of data stream bench mark known as The Linear Road bench mark. This paper also highlights 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 outlines the Linear Road bench mark. Section 5 discusses some of the commercial and prototype languages and systems. Section 6 provides a detailed discussion of the outlined data stream management system in regards to the technology and features which match and satisfy the Linear Road bench mark requirement, and Section 7 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 stream intensive applications, as it could not handle issues such as scalability, optimization and processing continuous real-time data.

Traditional Data Base Management System (DBMS) deals with stored sets of relatively static records with no pre-defined notation of time, while Data Stream Management Systems (DSMS) support online analysis of rapidly changing data streams. A data stream is defined as real-time continuous sequence of data which is too large to store; it is implicitly ordered by arrival time or explicitly by time stamp; queries in DBMS represented as one-time (transient) queries were in DSMS represented as continuous (persistent) queries. (Daniel J. Abadi & Michael Stonebraker) identified common requirements for DSMS (see Table 1 for more details of comparison requirements for DBMS and DSMS)

Data Base Management System DBMS	Data Stream Management System DSMS
1.Data processing results issuing transactions and queries 2.Manages data in its tables 3.Provide exact answers to exact queries and is blind to real-time deadlines 4.Optimizations of all queries in the same way 5.The norm is pull-based queries	1.Monitoring and alerting humans for abnormal activities 2.Processing of data that is bounded by finite window of values and not over unbounded past 3.Respond to real-time deadlines and provide reasonable approximations to queries 4.Benefits from Application specific optimization criteria Quality of Service (QoS) 5.The norm is push-based data processing

Table 1 DBMS and DSMS requirement

Stream processing approaches have been implemented in several stream oriented applications, including telecom call-records, network security, financial applications, sensor networks, web usage logs, and manufacturing processes.

In sensor networks, data stream processing has been used for highway congestion monitoring, geophysical monitoring, medical monitoring of life signs, energy resource monitoring and allocation, and supervision of manufacturing process. These applications involve complex filtering and activation of an alarm upon discovering unusual patterns in the data. Aggregation and joins over multiple streams are required to analyze data from many sources (sensors), while aggregation over a single stream required to compensate failure of single source (sensor) due failure of transmitting of potential data. Therefore, Sensor network applications may require access to pervious sensor stored historical data.

A representative sensor network in power station (Y. Chen, Dong, Han, et al., 2002) will have the following query processing capability:

- Activating a trigger of several sensors in the same area to report measurements that exceed a given threshold (for example if more than one sensor station exceeds the allocated amount).
- Drawing temperature contours on a weather map: by performing a join of temperature streams produced by weather monitoring station. Join the results with static table containing the latitude and longitude of each station, and connect all points that have reported the same temperature with lines.
- Analyze a stream of recent power usage statistics reported to a power station and adjust the power generation rate if necessary.

In network traffic analysis systems internet traffic is analyzed on real-time basis (Cranor, Gao, Johnson, et al., 2002). A network traffic analysis system requires features such as joining data different sources, packet monitoring, packet filtering, and detecting unusual condition and activities (denial or load of service). It also requires support for querying for historical data to compare current traffic traces with stored patterns corresponding to known events such as a denial of service attack, furthermore, other requirements such as monitoring recent requests to find the customers who consume the most bandwidth. For an Internet Service Provider it is important to distribute the bandwidth equally amount its users and preventing the case of a considerable amount of bandwidth being consumed by a small set of heavy users. Network traffic analysis uses the following queries:

- Traffic matrices: determine the total amount of bandwidth used by source destination pair grouped by distinct IP address, subnet mask, and protocol type. The IP traffic is statistically multiplexed, therefore a traffic stream must be logically de multiplexed in order to reconstruct the underlying TCP/IP sessions (Cranor, Gao, Johnson, et al., 2002). Dividing the stream into sessions involves temporal semantics (a session ends if two nodes have not sent packets to each other for some time).
- Compare the number of distinct source -destination pairs in the streams. If the counts differ by a large margin, then denial of service attack may take place and permissions to connect are not being acknowledged by the clients.

In online financial analysis an application analyzing stock prices involves discovering correlations, identifying trends and controlling opportunities, and forecasting future values. Any typical real-time financial application, allows users to pose queries such as:

- Finding all stocks priced between $100 and $150, where the spread between the high tick and low tick over the past 30 minutes is greater than three percent of the last price, and where in the last five minutes the average has surged by more than 300%.

- Finding all stocks where prices are within two percent of their respective 52-week highs that trade at least one million shares per day.

A number of requirements must be met by any given system in order for it to be considered as a stream processing engine. These requirements include high availability, scalability, and optimizations. As well as the desired capabilities which includes real-time computation on the data contained within an event, and the ability to detect and possibly react to simple and complex events instantly is also important.

A typical simulation prototype for DSMS known as the Linear Road Benchmark (A Arasu, Cherniack, Galvez, et al., 2004) developed by Aurora team and STREAM team, was described 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 coordination of the vehicle broadcasted 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 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, Amini, Andrade, et al., 2006). These features stretch the capabilities of the traditional data processing 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 traditional historical data.

Other past and current projects related to data stream management incude: The *XFilter* content based filtering system (Altinel & Franklin, 2000) performs efficient filtering of XML documents based on user profiles expressed as continuous queries in the *XPath* language . *Xyleme* (Nguyen, Abiteboul, Cobena, et al., 2001) is a similar content-based filtering system that enables very high throughput with a restricted query language. The *NiagaraCQ* (J. Chen, DeWitt, Tian, et al., 2000) system supports continuous queries for monitoring persistent data sets spread over a wide-area network, e.g., web sites over the Internet. NiagaraCQ addresses scalability in a number of queries by proposing techniques for grouping continuous queries for efficient evaluation.

3 Issues / Features of Stream Languages

Data Stream Management Systems (DSMS) are different from those of the Human Driven Systems (HDS) (Palmer, 2005) that exist today. An HDS requires user's interactions as it responds to information from human driven systems; while a DSMS pushes information to the user. A HDS has seconds, or even hours, to compute a request and come up with answers; while a DSMS has milliseconds to decide and come up with acceptable answers. An HDS is designed around daily, monthly or quarterly reports; while a DSMS provides dashboards that allow users to constantly analyze operations in real time or automate the actions that it discovers.

In DSMS Event Stream Processing (ESP) deals with the task of processing multiple streams of event data with the goal of identifying the meaningful events within those streams with almost no latency within milliseconds response. The ESP job is to consume multiple streams of event oriented data, analyze those events to discover patterns and act on incidental events it finds. The most critical component of an ESP platform is an event processing engine designed to process thousands of events a second with a capability of executing thousands of active event rules in real-time.

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 past literature and 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 decreases 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 (see Figure 1(StreamBase)). 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.

A common need for many stream processing applications is the ability to perform on real-time basis, where the result is produced in minimal delay and with a high throughput rate.

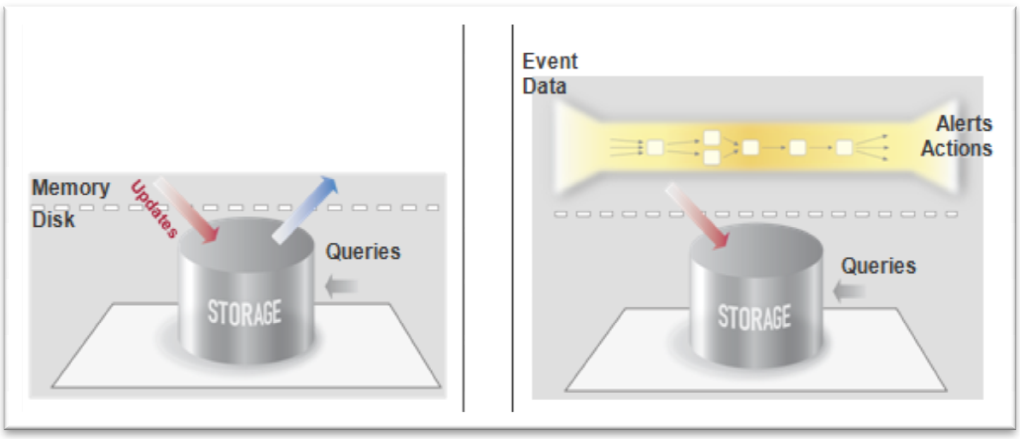

Figure 1 – traditional processing vs. stream processing

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 represented and stored in the language variables making it difficult to share among various applications. The physical data independency implies hiding the underlying details of how the data is physically represented and managed by providing a restricted interface, which will enable the system to easily modify the internal data structures without modifying existing applications. The logical data independency implies allowing different application to have its own view of the data.

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 (A 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, Balazinska, Rasin, et al., 2003).

Streaming data manipulated in three different querying paradigms:

- **Relational-based:** a declarative query language with SQL like syntax and enhanced support for windows and ordering such as Continuous Query Language (CQL)(Arvind Arasu, Babu, & Widom, 2003) from STREAM, and Continuously Adaptive Continuous Queries (CACQ)(Madden, Shah, Hellerstein, et al., 2002) from TelegraphCQ). The relational based model includes sliding windows and operators that translate relation to streams with possibility of specifying width of window.

- **Object-based:** a declarative query language with object oriented stream modeling, it classifies stream elements according to hierarchy type, such as (COUGAR (Yao & Gehrke, 2002))

- **Procedural:** the procedural model is an alternative to declarative query languages where users can specify the data flow and construct query plans via a graphical interface (Aurora(see section 4.1) is an example of procedural model)

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, Ahmad, Balazinska, Çetintemel, Cherniack, Hwang, Lindner, Maskey, Rasin, et al., 2005, p.4)

The language should accommodate query plans that uses non-blocking operators by using blocking stream operators, as well as using aproximation techniques to resolve data stream imperfections, any streaming operation can be unblocked by restricting its range to a certain finite window to fit memory size, to support incremental computation of operators over streams.

Three general techniques used for unblocking stream operators windowing, incremental evaluation, and exploting stream constraints (Golab & Özsu, 2003). If none of the three unblocking conditions are satisfied, compact stream summaries maybe stored with approximate queries created over the summaries, which will imply trade-off between accuracy and amount of memory used to store stream summaries; the approximate methods could be used to resolve this issue (see section 3.4).

Techniques such as dynamic revision of query result implemented in some of DSMS such as borealis (DJ Abadi, Ahmad, Balazinska, Çetintemel, Cherniack, Hwang, Lindner, Maskey, Tatbul, et al., 2005), will enable stream originator to correct errors in previous data sent, errors encountered due the unpredictable nature of stream source such as (data missing its processing window, data ignored temporarily due overload situation, and late arrival of data), in these cases the user is forced to live with imperfect results, unless there is a mechanism to revise the past process taking into account newly available updated data.

3.4 Providing Predictable Output

A stream processing system should provide a built-in mechanism and operators, to process minimal set of the received data (incomplete data stream for given operation), which may require stream processing languages to support single pass algorithms over data, 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, p 5). Another example is MAIDS (Cai, Clutter, Pape, et al., 2004). MAIDS 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.

Online stream mining operators must be incrementally updatable without making multiple passes over the data; recent results in approximate algorithms for stream mining(Golab & Özsu, 2003) include computing stream signatures and representative trends, decision trees , forecasting , k-medians clustering, nearest neighbor queries, and regression analysis.

Approximation techniques used to handle loads (streams coming too fast), avoid unbounded storage computation, and for queries that require approximate history, several techniques and methods could be used for approximating and data reduction such as:

- **Counting methods:** which used to compute frequent item sets and store frequent counts of selected item type.
- **Sampling methods:** compute various aggregates within a known error bound.
- **Sketches:** used to as summaries of stream that occupy small amount of memory (for example randomized sketching).
- **Wavelet:** reduce the underlying signal to a small set of coefficients to approximate aggregates over infinite stream.
- **histograms:** approximate frequency of element values in stream.

- **load shedding**: drouping random low priority tuple using technique based on selecting random sampling operations into query plans, to reduce and free resoucre overhead (Brian Babcock, Datar, & Motwani, 2004)

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. Another example is when dealing with financial stock market data to produce real-time based result based on historical and stream data (see example 1, the example demonstrates a query that represents dealing with stored and real-time data). The system should use standard 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.

Consider two stock data feeds, one containing TICKS data with fields such as (stock_symbol, volume, price, time) and other SPLITS feed with fields such as (symbol, time split-factor) which indicates when a stock splits.

To produce the split-adjusted price for security in a feed over several days it requires to access both stored and stream data in same manner

Stored table:	Store (symbol, factor)
Feeds:	Tick and Split

```
UPDATE Store
      (SET factor = factor * S.split_factor)
      FROM Split S
      WHERE symbol = S.symbol

SELECT T.symbol, price = T.price * S.factor,
       T.volume, T.date, T.time
FROM Tick T, Store S
WHERE S.symbol = T.symbol
```

Example 1: using stored and stream data in query execution

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 DSMS 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 number of techniques addressing these issues (Balazinska, Hwang, & Shah). Addressing this issue will enable the DSMS to handle failures and fault tolerance to enable high availability.

Processing stream data require long running queries on variable system conditions and unpredictable environment which requires special techniques to free the unbounded memory and processing power. Monitoring high availability and query distribution uses techniques such as cost metrics measurement to monitor accuracy and report delay vs. memory usage, output rate, and processing power usage, which then could be used by query optimization techniques based on resources, stream rate, and Quality of Service QoS. Queries can be rewritten to minimize cost metric adaptive query plans, due changing processing time of operators, selectivity of predicates and stream arrival rates

To maximize high availability and provide fault tolerance some of the current stream processing engines, use a replication of running multiple copies of the same query over networked distributed Stream Processing Engine (SPE) nodes, so that when a node detects a failure on its input stream, it will switch to alternate stream replica to continue processing.

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, Ahmad, Balazinska, Çetintemel, Cherniack, Hwang, Lindner, Maskey, Rasin, et al., 2005p. 7) for 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 rewrite low level codes.

Scalability involves sharing execution of many continuous queries, monitoring multiple streams, reduction of tuples through several layered operations (several DSMSs) distributed DSMS, adoption of per element processing single pass to reduce drops, blocking processing to optimize I/O cost.

Query operators and optimization is another concept which needs to be taken into account in order to enhance scalability and resource utilization; operators are categorized as stateless that do not need to maintain internal state such as (selection operator and the projection operator), and stateful operators that need to maintain the internal state such as (join and aggregate operators). The challenges to implement these operators and to execute the queries to provide scalability, the operator execution may be shared between different concurrent queries. The need for a real-time capability, algorithms may require restricted single pass over the data.

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, Diao, & Rizvi, 2006). Event processing refers to techniques such as filtering, correlating, aggregating, detecting complex patterns of many events and relations between events to be computed on real-time bases. 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.

Stream processing data similar to traditional active base-like event processing as streams are equivalent to (time ordered) event history and continuous queries are equivalent to event expressions, which continuously monitor the event history for matching patterns.

Most of the current stream processors use SQL-like languages with windows over stream extension for expressing continuous queries; computation over streams differs from traditional relations as streams are continuous and potentially infinite. This limits the range of operators that can be applied to streams to only the set of non-blocking operators, as the result will not wait for the end of input data, which implies some of window blocking operators could not be performed over a stream such as sorting.

The use of semantic windows described by (Rizvi, 2005) goes beyond time-based or tuple-based windows as they not defined based on a number of tuples or a time interval but rather semantically, based on the occurrence of complex events, for example, consider a query that wants to find the average temperature in the home between the time a person comes home and when he goes out. This query is looking at a window over a temperature sensor stream that starts and ends based on “person" events.

Real-time reactions for monitoring applications, such as active mechanisms are very important concepts in dealing with events processing based on their accordance. (Roberto, Max, & David, 2002) described Distributed Event Awareness Language DEAL as an event processing language and system designed to provide awareness information in a heterogeneous distributed system. The DEAL environment extends the basic functionality provided by even notification servers such as khronika, CASSIUS and ELVIN to cope with the richer set of requirements of awareness applications; this is accomplished by the use of a powerful event language that allows the definition, processing and combination of filtering and routing of events coming from heterogeneous sources.

4 Linear road bench mark

The Linear Road endorsed as a benchmark for Data Stream Management System DSMS("Linear Road,"), is designed and developed by both Aurora Team and STREAM team. The bench mark system makes it possible to compare performance characteristics of different Data Stream Management Systems relative to each other. The Linear Road (A Arasu, Cherniack, Galvez, et al., 2004) is designed to test and measure the maximum scale of given sets of both continuous data streams and historical data at which any given DSMS can respond to, by measuring how many expressways a DSMS can support by given accurate and timely results to four types of queries that fit two categories (continuous and historical data).

4.1 Linear Road query requirements

Linear Road simulates a variable tolling system on expressways in a metropolitan area. Features such as accident detection and alerts, measurements of the traffic congestion, toll calculations, and historical queries from simulated drivers on the expressways are incorporated into the simulation. Certain criteria are also specified, such as the ability to maintain historical data and produce calculations on live data to produce results. The Linear Road bench mark based on the following four queries:

4.1.1 Accident Notification:

This query is based on accident detection and accident notification. An accident is detected when one or more vehicles stopped having a zero kilometers per hour at the same physical position at the same time. A vehicle is considered as stopped when it reports the same position continuously for four times over any given segment of the road. Once an accident is detected, every vehicle that enters into a segment of the road where the accident occurred must be notified on real-time basis so that it gives the opportunity for these vehicles to exit the expressway thereby avoiding the congestion. The accident notification query is based on continuous real-time query.

4.1.2 Toll notification:

Toll notification informs the vehicle toll access of the next segment of the toll road; it is triggered by a report of the vehicle position. It is based on continuous query as the results are constantly updated based on the contents of data such as the position of the vehicle. The toll processing is calculated when every time a vehicle reports its position in a new segment, and notifies the driver of this toll. The toll calculations are determined based on the current congestion on the segment of the road; the congestion of the given segment of the road is measured based on the number of vehicles and the average speed for the vehicles for given segment of the road.

4.1.3 Account balance:

This provides the total amount of balance for an account for a given vehicle in any particular moment. This query is based on historical data stored in the database. The linear road requirement for answering account balance queries means that the tolls charged to each vehicle must be maintained in a timely fashion.

4.1.4 Travel time estimation:

This is a request for an estimated toll and travel time for a trip on a given expressway based on date and time.

4.2 Bench mark requirement challenges

In stream data environment, streaming data poses the following challenges to the requirement of the benchmark. In order to test any data stream management system on the linear road bench mark, a DSMS needs to accommodate the following challenges:

4.2.1 Semantically valid input:

The input data to stream benchmark should not be based on random inputs but should have some semantic validity; the content of stream should be consistent with the activity for example considering implementing soldiers movement tracking application, if the soldier movement is reported every 10 minutes, the positions of two consecutive reports should not differ by more than how far the soldier can travel in that time.

4.2.2 Continuous query performance metrics:

Stream queries are mostly continuous, therefore the typical database standard metric which relays on completion time for an computation process is inappropriate for given queries since such queries are on continuous basis and never get completed, therefore such queries should use more appropriate form of measurement such as “response time” (to determine the average time for data when it enters and exit the DSMS as a computed response) and the “support of query load” (how much of input load a data stream management system can process while still meeting specific response time and constraints).

4.2.3 Many correct results:

Continuous query results may depend on developing historical state or the arrival order of tuples on a stream, therefore several different results for the same query may be correct. Validation of results should take into account the fact that some queries may have several correct answers.

5 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.

5.1 Aurora and Borealis

Aurora (D Abadi, Carney, Çetintemel, et al., 2003) is a general purpose DSMS designed by (Brandies University , Brown University and Massachusetts Institute of Technology). It is based on the data-flow 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 delivered to the corresponding nodes. Aurora can also maintain historical storage in order to support ad hoc queries.

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 operations employ `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, the `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.

The Quality of Service QoS data structures of Aurora attempt to maximize the perceived QoS for the outputs it produces; The QoS in general is a multidimensional function of several attributes of an Aurora system which includes:

- Response times- produces output in a timely fashion, as otherwise QoS will degrade as delays get longer.
- Tuple drops – if tuples are dropped to shed load, then the QoS of the affected outputs will deteriorate.
- Values produced – QoS clearly depends on whether or not important values are being produced.

The Aurora Storage Manager (ASM) is responsible of storing tuples required by an Aurora network; there are two types of necessities for the storage manager. Firstly to manage storage for tuples being passed through an Aurora network, secondly to maintain extra tuple storage that may be required at any given connection point.

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 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.

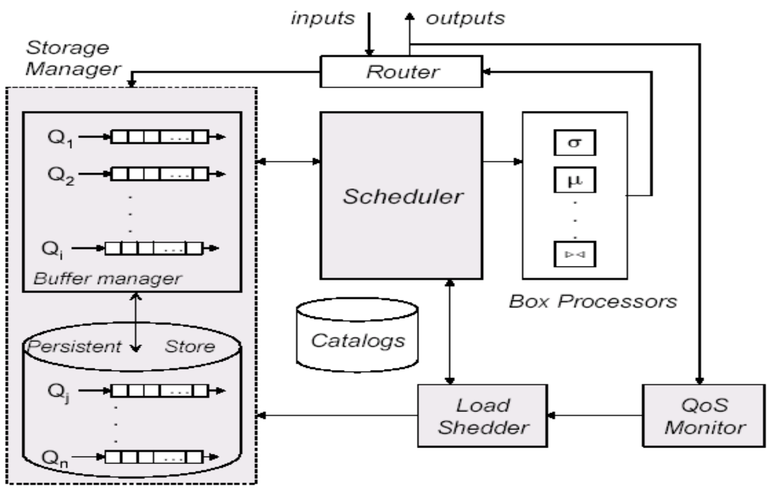

Figure 2: Aurora Architecture (component) (Aurora)

The Aurora system supports three types of queries:

- **Continuous query**: it accepts an input stream of data, and outputs the computed value to a listening application; the data need not be stored and discarded (for example: notifying the application whenever a continuously uprising of stock price for 1 hour is detected).
- **View query**: it is similar to continuous query, except that output data is stored temporarily for a prescribed period of time allowing the application to retrieve it later.
- **Ad-hoc query**: Handles query submitted to the system at any time (for example, allow the application to query the position of an object in the last 1 hour).

Connection points are places where boxes can join in the stream, at the connection point; data are stored temporarily in query and managed by the storage manager.

The Aurora system has seven primitive operators (known in Aurora terminology as boxes). These boxes are divided to two categories. The first type is known as windowed operators (order-sensitive operators). It is constrained by a time stamp such as (BSort, Aggregate, join, and resample). These operators require order specification arguments that describe the tuple arrival order. The second type is known as non-windowed (ordered agnostic) operators, that do not rely on time stamp and can always process tuples in the order they arrive, such operators include (filter, map, and join). The seven operators of Aurora are listed below:

- **Filter:** This operator allows tuples matching certain condition to pass through; it acts like a case statement and can be used to route input tuples to alternative streams. The output tuple has the same schema and values as input tuples, including Quality of Service QoS timestamps.

- **Map:** This is a generalized projection operator, the resulting stream has a different schema than the input stream, but the timestamps of input tuples are preserved in corresponding output tuples.

- **Union:** It is used to merge two or more streams into a single output stream; the union outputs all iprocessed tuples having the same schema and values as input tuples including their Quality of service time stamp.

- **BSort:** This is an approximate sort operator; since a complete sort is not possible over an infinite stream because of having infinite time, BSort performs a buffer based approximate sort equivalent to number of passes of a bubble sort. The BSort operator outputs exactly the same tuples in process with having different order on the tuples, the output tuples have the same schema and values as input tuples, including the quality of service timestamps.

- **Aggregate:** This operator applies window functions to sliding windows over its input stream.

- **Join:** This operator combine tuples from two or more streams if the difference in time stamp is no more than a threshold

Resample: This operator interpolates tuples between actual tuples coming from the stream; it is a semijoin like synchronization operator that can be used to align pairs of streams

The Aurora shedding is also related to approximate query answering, data reduction, and summary techniques, where result accuracy is traded for efficiency. By throwing away data, Aurora bases its computations on sampled data, effectively producing approximate answers using data sampling. This is a unique aspect of Aurora derived by quality of service QoS specification.

Borealis (DJ Abadi, Ahmad, Balazinska, Çetintemel, Cherniack, Hwang, Lindner, Maskey, Rasin, 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, Balakrishnan, Salz, et al., 2003) .

Borealis can be seen as a giant network of operators (aka query diagram) whose processing power is distributed to multiple sites. Sensors networks can also participate in query processing behind a sensor proxy interface which acts as a replica of Borealis site. Each borealis node server contains Query Processor (QP) which forms the local server processor. Input streams are fed into the QP and the result fetched by Input / Output Queues. The I/O Queues responsible of routing tuples from and to other Borealis nodes and clients. The Admin module within Borealis is responsible of controlling the QP by moving query diagram fragments from and to remote Borealis nodes, on the same time the admin module is broadcasting messages to local optimizer whose communicates further with major run-time components of the QP to extract performance improvement directions. The major components of QP are:

- Priority scheduler: It determines the order of box execution based on priorities of a given tuple.
- Box processor: each different box has a single box processor which can respond to changes of behavior on the fly based on control messages from the local optimizer.
- Load Shedder: it is responsible for eliminating low priority tuples when a given node is overloaded.

The Query Processor also contains Storage Manager, which is responsible for storage and retrieval of data flows; another component of QP is Local Catalog which stores the meta-data and description of query diagrams. The Neighborhood Optimizer uses local load information as well as from other Borealis nodes to improve local processing balance between the borealis connected nodes.

Unlike Aurora, Borealis has improved High Availability (HA) modules hosted on the nodes of Borealis; the High Availability modules monitor each other and enhance load balance across the nodes in case of node failure. The Local Monitor collects performance related statistics as the local system runs to report to local and neighborhood optimize modules.

Borealis allows Quality of Service QoS to be predicted at any point in a data flow. Therefore messages are supplied with a vector of metrics. These metrics include content related properties.

Aurora applies an append only model in which a tuple cannot be updated once it is placed on a stream which may lead to imperfect results due to using approximation techniques, while in Borealis this capability is extended by using dynamic revision of query results by allowing tuples to be updated and at any given time. The goal is to process revisions intelligently, by correcting query results that have been produced in the past in a more consistent way.

Borealis enhanced Aurora’s optimization to support three issues, firstly, it intends to optimize processing across a combined sensor and server network. Secondly to deal with Quality of Service metric as it is an important for stream based applications. Finally, to enable in dealing with large volumes of data generated by multiple distributed data sources.

An example of using SQuAl Query in Aurora is the use for military logistics applications. The inputs for the query are streams of soldier positions reported with the schema Sid for soldier identification, time and pos for soldier position. Having a query such as “produce an output whenever m soldiers cross some border k at the same time (with border crossing detection determined by the predicate Pos >=k). The SQuAl expression for this query requires, first to filter position reports for those that reveal that a soldier is across the border. Then, an aggregation on the resulting stream (assuming an order on time) produces a count of soldiers reporting a position beyond border k at each point in time. Finally, a filter identifies those times where the soldier count exceeds m.

Considering another example to “compute the center of mass of every soldier’s position for every timestamp. Alert when it differs from previous reading by 100m. Assuming that it is not worth waiting for more than t seconds for soldier position reports when calculating center of mass for a given time”. The SQuAl expression for this query requires, first to calculate the center of mass (using a user defined function) over all position reports based on time. A timeout argument ensures that a center of mass calculation never requires waiting more than t seconds from the time of the first soldier reports to the last. Secondly an aggregate box (operator) maintains a window of size two tuples, so as to compare the most recent two center of mass calculation to see if they differ by 100 or more. The user defined window function performs the calculation and outputs a flag which is set to TURE if the last center of mass calculation differs by the previous one by 10m or more, Finally a filter box (operator) is applied to return the center of mass calculation and the associated timestamps for those having the flag set.

5.2 STREAM

STREAM (Arasu A, Babcock B, Babu B, et al., 2003) stand for Stanford stream data manager, is a general purpose centralized single system Data Stream Management System DSMS produced by Stanford University. It supports large declarative continuous queries over continuous streams and traditional data sets. STREAM is built on the basis of two fundamental differences than traditional DBMS, firstly it must handle multiple continuous, high-volume, and possibly time-varying data streams in addition to managing traditional stored relations. Secondly it should support long-running continuous queries and produce answers continuously in a timely fashion.

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. The 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.

Continuous Query Language CQL (Arvind Arasu, Babu, & Widom, 2003) is a declarative query language derived from the SQL language with respect to the stream processing requirement (See addressed issues and challenges (B Babcock, Babu, Datar, et al., 2002). A registered CQL in STREAM system produces a complied 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. The abstract semantics implementation uses three types of operators over stream and relations `relation to relation`: uses SQL constructs to express its relation to relation operators, `stream to relation`: based on a concept of sliding window over stream expressed using window specification language derived from SQL-99, and `relation to stream`: has three operators, while the stream to stream manipulation 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.

STREAM has two data types in the query language: stream, defined as bags of tuple-timestamp pairs, and relations defined as time-varying bags of tuples. Both of these types are additionally flagged with either an insertion or deletion. The streams only include insertion, while relations may include both insertion and deletions used to capture the changing relation state over time.

Queries in STREAM are executed in Query plan; a scheduler selects operators in the plan to execute in turn. The semantics of each operator depends only on the time stamp of the elements it processes, not on the system time. Therefore, the order of execution has no effect on the data in the query result; however, it may affect other properties such as latency and resource utilization.

STREAM system has several operators used to implement the CQL language to deal with both data types (stream and relational):

Relation to stream operators:

- **Select:** Filters elements based on predicate.
- **Project:** Duplicates preserving projection.
- **Binary-join**: Joins two input relations.
- **mjoin:** implements Multiway join.
- **Union:** implements bag union.

- **Except:** implements bag difference.
- **Intersect:** implements bag intersection.
- **Antisemijoin:** Antisemijoin of two input relations.
- **Aggregate:** Performs grouping and aggregation
- **Duplicate-eliminate**: Performs duplicate elimination

Stream to relation operator:

Seq-window: Implements time-based, tuple-based, and partitioned windows

Relation to stream operators:

- **I-stream**: Implements *Istream* semantics
- **D-stream:** Implements *Dstream* semantics
- **R-stream:** Implements *Rstream* semantics

Additionally there are several system operators to handle housekeeping tasks such as marshaling input and output and connecting query plan together. During execution, operators are scheduled individually, to allow control over queue size and query latencies.

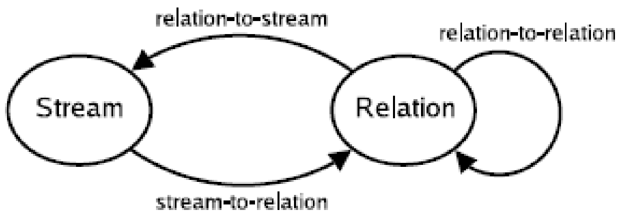

Figure 3: Data types in STREAM (STREAM)

The STREAM (A Arasu, Babcock, Babu, et al., 2004) system includes a monitoring and adaptive query processing infrastructure called `StreaMon` which monitors the performance over a time as query loads and system conditions change;, 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.

SPREAD is a stream to stream operator; its goal is to create and manipulate ordering relationships among the tuples (Zdonik, Jain, Mishra, et al., 2008). It creates a new equivalence relation that is a refinement not only of the timestamp equivalence relation but of a possibly existing batch equivalence relation on the input stream.

An Example of CQL Queries is given by the following Query:

Select Istream (*) From S [Rows Unbounded] Where S.A >10

The stream S in the example is a converted into a relation by applying unbounded (append-only) window. The S.A is a relation to relation which acts over the relation, and the inserts to the filtered relation are streamed as the result.

Here is another example of CQL which joins two streams:

Select * From S1 [Rows 1000], S2 [Range 2 minutes] Where S2.A =S1.A And S1.A > 10

The answer to the above query is a relation; at any given time, the answer relation contains the join (on attribute A with A >10) of the last 1000 tuples of S1 with the tuples of S2 that have arrived in previous 2 minutes. To produce a stream containing new A values as they appear in the join, the query needs to be rewritten and "*" in select clause is changed to "Istream(S1.A)".

5.3 StreamBase

StreamBase (2009) is a commercial Data Stream Management System 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 implementations 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 value acquired by evaluating the window on the history of input streams of that tuple.

StreamSQL is graphical event flow programming language which extends SQL and composed of several operators to allow processing of real-time data streams and historical data; the Stream SQL manages continuous event stream and time-based records, it retains the capabilities of SQL while adding new capabilities such as rich windowing system and the ability to mix stored data with streams. StreamSQL extends SQL in `Data Windows`: which defines scope of an operator over time, `integrated Access to Stream and Stored data`: which handles manipulation of both stream and historical data in uniform approach, and `Stream specific operators and constructs`: which offers set of stream-oriented operator which allows temporal pattern matching over stream and manipulation of stream data. The operators provide capability of filtering, merging and combining of streams as well as running time window based aggregations and computations on stream, furthermore it handles disordered, late and missing data.

StreamBase provides a palette of data processing operators, each of which performs a specific run time action on streaming data. The operators used in StreamBase are listed below:

- **Aggregate:** provides a view into data in a moving window, it can be used to compare, sum average, or count the data.

- **BSort:** uses an approximate buffered sort to record tuples, it is used to reorder records by time where there are a large number of out of order data.

- **Filter:** used to send tuples to different downstream branches based on tuple values, and also used to remove tuples from streaming data where the data of those tuples do not correspond to a defined set of values.

- **Gather:** combines tuples, by using common key value, it also recombines tuples that previously split up into separate branches.

- **Join:** combine tuples where there can be more than one match per tuples.

- **Lock and unlock:** used to process on tuple at a time through a section of application to achieve mutual execution. The operator used to ensure sequence coordination, or to protect shared data against concurrent writes, within the set of operators that reside between the starting lock operator and the ending unlock operator.

- **Map:** transforms tuples by performing math or logical functions, or remove or rename fields in a schema.

- **Merge:** combine two streams into one with ordering by value (such as time).

- **Metronome:** delivers output tuples periodically based on the system clock, at custom specified interval.

- **Query:** the query operator reads and writes to query table or external RDBMS through JDBC. The query table can reside in memory or on disk. The query operator and query tables enable sharing of data state across a StreamBase application.

- **Split:** controls the order in which the outputs of a multi output operator should be processed to completion by each set of downstream component. The split operator explicitly controls the downstream processing sequence.

- **Union:** combine two streams with the same schema into one stream, with no particular interleaving as in merge.

- **Heartbeat:** The heartbeat operator triggers actions based on timing information contained in a stream, to ensure that downstream operators are not disrupted by late or missing data.

StreamBase also allows additional custom operators which can be used to implement customized functionality by using interfaces and adaptors to external sources. The adaptors are used in converting streaming data into the protocol required by StreamBase and vice versa.

StreamBase also provides a standard interface to most modern data management systems through its JDBC interface.

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 range of adapters and interfaces which enable the conversion of streaming data to required procedures by StreamBase.

To preserve the integrity of mission critical information and avoid disruptions in real time processing, the high availability in StreamBase is addressed by providing standard process pairs approach of two dedicated servers, one as primary and the other as a backup using specific mechanism which enables asynchronous synchronization to take place and prevent overhead accordance

StreamBase is designed to scale up from a single machine implementation to a large distributed multi server deployment. In addition it offers higher scalability, and handling of unexpectedly high peak loads processing could be transparently distributed to a cluster of machines across the network.

An example for using StreamBase is the implementation of Binary Large Object BLOB used to process streams of image files, frames from a video camera pushed individually into the application as a part of StreamBase tuple. The pre defined StreamBase operators perform some initial image analysis, and add the results to fields in the tuple. The downstream components in the application would then use this metadata to make decisions on how to further process the tuples. For example, a function might extract the header from a BLOB in the image format, storing the required information (such as image resolution) in the tuple field. Considering a video stream example, where police cameras may be sending video frames from stoplights throughout a city, the application requires processing these frames in a series of stages. First, by performing pre-analysis to determine whether or not there was a readable license plate in the frame, the initial function add an indicator of this to the other metadata fields in the tuple (such as location, time and camera number). The next operator on the process is to make a decision about the tuple by looking at the new field in the tuple. If there was no license plate detected, it would discard the image. If it detects a license plate, it would then compare the plate to a database of those being actively sought (stolen cars). If a match was found, it would then initiate the appropriate follow up response and pass it to downstream application for more processing.

5.4 SPADE

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

SPADE leverages existing infrastructure of stream processing core SPC provided by System S; it utilizes code generation framework to create highly optimized application that 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 that 100 physical nodes in a strongly connected cluster environment. The system is expressed as a dataflow graph consisting of processing elements (PEs) that consume and produce streams of data through input and output ports; the main components of SPC are `Dataflow Graph Manager:` responsible for determining connections among PEs and matches stream descriptions between input and output ports, `data fabric:` establishes transport connections between PEs and moving data streams from producer to consumer as well as choosing appropriate transportation to achieve flow balancing among PEs to ensure stable operation within workloads, `Resource Manager:` makes global resource allocation decisions by sharing system infrastructure and `PE execution container:` provides runtime content and access to SPC middleware also monitors resource usage.

SPADE's syntax and structure are centered on exposing the controls to the main tasks associated with designing application; it effectively hides the complexities related to stream manipulation (generic language support of data types and building block operations) and also supports application decomposition in a distributed environment.

SPADE deals with incoming data from external sources either by converting the data source into a data flow using SPADE's source operator which interprets the data to System S format, or by performing data transformation on the incoming stream, carried out by another SPADE language operator known as a user-defined operator.

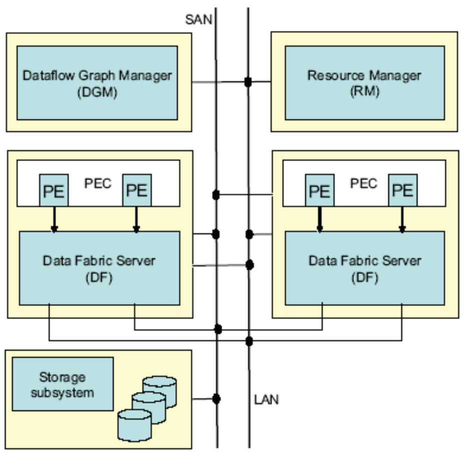

Figure 4: The Stream Processing Core

SPADE provides set of stream-relational operators, which can be used to implement any relational query as well as windowing extensions used in streaming applications. The supported SPADE operators include:

- **functor:** a functor operator is used for manipulating tuple-level (filtering, projection, mapping, attribute creation and transformation, the functor operator can access past tuples that appeared earlier in the input stream.
- **Aggregate:** this operator used to group and summarize the incoming tuples, it support large number of grouping mechanisms and summarization functions
- **Join:** used to correlate two steams, streams can be joined in several ways.
- **Sort:** A sort operator is used to force order on incoming tuples in a stream using several methods
- **Barrier:** A barrier operator is used to synchronize tuples from multiple streams upon arrival of tuples from input stream.
- **punctor:** A punctor operator is used to manipulate tuple-level where condition of both current tuple and past tuple are evaluated for generating punctuations in the output stream
- **Split:** A split operator used for routing incoming tuples to different output streams based on a supplied condition.
- **Delay:** is an operator used for delaying a stream based on pre-defined time interval.

SPADE contains an edge adapter (Sink and source) which is described as language operators. The source operator is used to create a stream from data coming from external source, and it is capable of performing parsing and tuple creation by interacting with range of external sources. The sink operator is used for converting stream into a flow of tuples that can be used by other external source; its main task is to convert tuples into externally accessible objects; it is used to write to external files and sockets

The SPADE language also provides the capability for extending the basic operators by supporting range of user-defined operators by providing access to external libraries and implement operators that are customized to a particular application domain.

SPADE 2 (Martin Hirzel, 2009) has added some extra features to SPADE 1 such as composite operators, shared variables, and richer data models as well as scaling the design to allow efficient distributed implementation. The primary goal of SPADE 2 is to permit efficient implementation on distributed hardware. The majority of changes aim to make the language more uniform and simpler.

An implementation example of SPADE application is from the finance domain, known as Bargain index computation (Gedik, Andrade, Wu, et al., 2008). Considering a stock trading scenario, where the aim is to find bargains to buy. The Bargain index is a scalar value representing the size of the bargain to compute the bargain index for every stock symbol that appears in the source stream. The visual representation of the SPADE query that implements the logic uses the SPADE built-in operators. The source data contains trade and quote information. A trade shows the recent traded stock price, while a quote could have values either “ask” price to sell a stock or a “bid” price to buy a stock. Computing the bargain index requires firstly processing the logic by separating the source stream into trade and quotes. This is achieved via using two functor operators. The functor operator creates the upper trade branch and it also computes trade price x volume, which will be used later to compute the volume weighted average price. The aggregate operator computes a moving sum over price x volume. It uses per-group window of size 15 tuples with a sliding of 1 tuple. It outputs a new aggregate every time it receives a trade. Another functor operator uses to divide the moving summation of price x volume to that of volume, giving the most recent volume weighted average price value of the stock symbol of the last received trade. The result of the stream is connected to the first input of an equi-join (on stock symbol) operator, which has a per group window of size 1 tuple on the same input. In other words, the join window for the first input has one group for each unique stock symbol stores the last value within each group. The second input of the join is connected to quote stream, and has a zero sized window (single sided join). The aim is to associate the last received quote with the most received volume weighted average price value computed for the stock symbol of that quote. Once the operation is completed, a simple formula is used to compute the bargain index as a function of the ask price, ask size and the volume weighted average price value. The final functor operator then filters out the bargain indexes that are zero to indicate a bargain has not been detected.

6 Discussion

Data stream management systems have been used in a wide range of applications, each domain having specific needs and requirements. In this section we outline the four discussed stream processing engines along with the technology and language used in each of the systems to map up with the requirements of the linear road benchmark.

An application such as the linear road bench mark needs the ability to handle a large number of stream data with minimal latency. We can note that all of the four systems discussed in the paper have the capability of dealing with stream data in low latency and high volume processing manner, as these four systems process data streams on the fly without costly storage operations. In addition latency problem exists with systems that are passive (systems that wait to be told what to do). StreamBase and SPADE systems highly accommodate this capability as they are both specifically designed to address the high volume data processing and low latency by processing the data on real-time basis. StreamBase architecture is primarily designed to minimize response time while supporting high data rate. SPADE also accommodates the full capacity of supporting low latency and high volume processing since it is designed based on the full features of the existing stream processing code (SPC) provided by system S. While both STREAM and Aurora can partially support low latency and high volume processing, Aurora and STREAM could possibly deal with high data volume and low latency as both are designed based on a single stream environment which may limit their capability to deal with massive incoming data. However, the STREAM engine enables high performance by sharing the state and computation across query plans as well as the use of techniques such as ordering and clustering which enhance resource usage. While Borealis (the next generation Aurora) could also possibly process high volumes of data streams with low latency as Aurora extended its single processing capability to a distributed environment to perform processing on multiple nodes.

The core principle of a stream processing application is to deal with streams of data as well as historical data (a repository data) as needed; all of the discussed stream processing management systems employs data independency by enabling the use of declarative languages such as SQL. The four stream processing management systems have an extended next generation SQL. Those extended next generation SQL have built in operators that handle the manipulation of stream data and historical data in the same manner, in addition the four systems support dealing and accessing stored data in one way or another. In the linear road there is a need to deal with two different types of data, for example, accident detection query requires dealing with real-time data while account balance requires accessing stored historical data. The Aurora system uses an extended SQL known as Stream Query Algebra SQuAl which incorporates seven well defined operators. Aurora uses two types of operator to deal with stream based data and non stream based data (historical data). The ordered tuple manipulation based operators are used to process historical data, while non-ordered tuple based operators are used to manipulate stream data. Similarly the STREAM system employ an extended SQL language known as Continuous Query Language CQL to manipulate stream and historical data. STREAM data manipulation is based on three categories: a relation to relation, a stream to relation and relation to stream. These three categories are supported by operators which perform the actual processing of the data. StreamBase data management system uses a variant of extended SQL language known as StreamSQL. StreamSQL is defined as an event flow programming language. It has its own set of operators designed specifically to deal with continuous stream and historical data. The SPADE System hides the complexities related to stream manipulation by using a generic language to support data types and building block operations. SPADE deals with stream data and historical data either by converting the data source to data flow by using SPADE's source operators or transforming the data by using SPADE language operators known as user-defined operators. SPADE has a set of stream-relations operators which are used in relational query and continuous data streams.

The nature of stream data is different than the traditional ones as it relies on continuous real-time data. There will be always cases where streams of data are incomplete (missing, delayed and out of ordered), for example in linear road the accident detection is determined by manipulating four current physical positions of a vehicle provided in row by a given vehicle sensor; having a missing or delayed of one of the positions due lack of communication will impact on the result of determining the accident occurrence. Such a situation requires the system to have special techniques to handle incomplete data and provide output by extending an opened processing window or ignoring a missing position data based on the next received position data. The four discussed data stream management systems have the ability to handle stream imperfections (delayed, missing and out-of-ordered data) using different techniques. The Aurora system uses slack and time-out parameters to deal with slow and out of ordered data; it also employs several Quality of Service graphs to enable better performance monitoring such as latency graph to deal with delayed result and value based graph in dealing with important results. The STREAM system uses techniques such as load shedding and dropping when data arrival rate is high and the computing exceeds the available memory. The StreamBase uses special operators which provide the capability of managing disordered streams and late or missing data; a heartbeat operator is used in StreamBase to trigger actions based on timing information to ensure the operation are not disrupted by late or missing data, it uses system clock and produces output every so often. Among the SPADE system user-defined operators, a functor operator is used to access past data (tuples that have appeared earlier in the input stream and a punctor operator is used to evaluate generation of out-of-bond (punctuations) which may rely on other user-defined operation such as sort, aggregate and join.

Due the nature of unbound continuous streams of data (an infinite continuous data, where data may not arrive), a stream application needs a special mechanism to enable the ability to provide predictable result based on special computation, in order to output final result and release processing power of the system. Most of the data stream management systems have a special technique to handle such a situation while others not, depending on the implementation domain. In the linear road, the application requires such techniques to enable providing approximation result in order to complete a processing element (for example a missing data from a sensor vehicle could be handling by predicting approximation on bases of some other past result or stored historical data). Aurora system uses loss tolerance graph to handle incomplete and approximation answers. In STREAM, the system handles overload situations when the load of execution exceeds the allocated memory size by providing approximation techniques such as load-shedding, sampling and dropping of unnecessary tuples. StreamBase processes tuples in time-series manner; it also employs techniques such as time-out and approximating results. The SPADE system does not have any implementation to predict results.

Data integration is another core element of data stream management systems, as such systems require seamless integration between continuous data and historical data having trouble-free access to both types, as discussed in section 5, The Linear Road requires access to both types in a similar manner. The discussed four systems have this feature taking into account each system has different techniques in implementing the integration of both types.

Most of current real-time applications are measured based on their reliability in being highly available and easily scalable, dealing with critical system such as the linear road; it is very important that such an application can serve and always be responsive, generally high availability in stream processing application is addressed in using techniques such as multi-processing on distrusted environment, resource sharing and utilization. The Borealis (next generation of Aurora) has implemented the use of distributed environment, which enables the processing over multiple nodes which makes high availability and scalability. The STREAM is designed as a single Data stream management system which makes it difficult to scale and be highly available. StreamBase addresses high availability by providing standard processing pairs of two detected servers. The SPADE system inherits scalability and high availability from the core implementation of system S.

The discussion illustrated four representative-stream processing engines along with the language and underlying technology features used to address the requirements of stream processing applications. Table 2 summarizes the results of these four systems presented in this paper along with those of the Linear Road benchmark.

	Linear Road	Aurora/ Borealis	STREAM	StreamBase	SPADE
Low-latency/high-volume	Yes	Yes	Possible	Yes	Yes
Data independency	Yes	Yes	Yes	Yes	Yes
Incomplete data stream	Yes	Yes	Yes	Yes	Yes
Predictable output	Yes	Yes	Yes	Yes	No
Data integration	Yes	Yes	Yes	Yes	Yes
High availably	Yes	Possible	No	Possible	Yes
Scalability / utilization	Yes	Possible	No	Possible	Yes
Complex event processing	Yes	Possible	Yes	Yes	Yes

Table 2: technology features

The cells in the table contain one value of three possible values. “Yes” indicates the technology supports this feature; “No” indicates the technology does not support this feature; and “Possible” for possibility of support with modifications and enhancement.

7 Conclusion

Stream processing applications have been known for several years, as they have emerged from different requirements of applications as discussed in section 2. There are a number of issues and features required for such an application which depends on many factors such as language features and system implementation.

Data Stream Management Systems DSMS's are different from those of the human-driven systems Human Driven Systems HDS as they process data without involvement of the user as they rely on occurrence time, causal relationships and event abstraction as their fundamental operations, rather than the structured data and relational algebra supported by a traditional database system.

Stream data will often have longer-term value as part of the data that drives stream processing language actions. Many Data Stream Engines (DSE) require rapid response as well as incorporating historical data as a part of the logic that drives the processing computations, therefore a stream processing language needs access not just to what just happened, but also what happened at an earlier time.

Since DSMS deals with high volumes of data streams from automated sources, where the sources can generate thousands of data streams per second. DSMS provides a new type of software platform that includes stream programming language, stream visualization tools, adapters for streaming data and an event database. A stream processing engine runs streaming data over events rules; by contrast, an SQL query processor executes queries against data set.

A stream processing engine scales by implementing a distributed processing architecture in which streams are handled in threads, with an optimizer deciding which stream of data is to be executed among thousands of active data streams. A DSMS employs pipelining techniques that are often associated with the machine hardware design.

A Data Stream Engine implements operator instructions it receives from stream processing languages; a stream processing language defines concise syntax that searches for and acts on patterns amongst multiple inbound event streams. The language design must support parallel query processing.

References

Abadi, D., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J., Lindner, W., Maskey, A., Rasin, A., & Ryvkina, E. (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., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J., Lindner, W., Maskey, A., Tatbul, N., & Xing, Y. (2005). *Design issues for second generation stream processing engines.* In.

Abadi, D., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Erwin, C., Galvez, E., Hatoun, M., Maskey, A., & Rasin, A. (2003). *Aurora: a data stream management system.* In 666-666,ACM New York, NY, USA.

Altinel, M., & Franklin, M. (2000). *Efficient filtering of XML documents for selective dissemination of information.* In 53-64.

Arasu A, Babcock B, Babu B, Datar M, Ito K, Motwani R, Nishizawa I, Srivastava U, Thomas D, Varma R, & Widom J (2003). *STREAM: The Stanford Stream Data Manager*. Retrieved from http://ilpubs.stanford.edu:8090/583/

Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., & Widom, J. (2004). Stream: The stanford data stream management system. *a book on data stream management edited by Garofalakis, Gehrke, and Rastogi*.

Arasu, A., Babu, S., & Widom, J. (2003). *The CQL Continuous Query Language: Semantic Foundations and Query Execution* (Technical Report): Stanford InfoLab.

Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A., Ryvkina, E., Stonebraker, M., & Tibbetts, R. (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.

Aurora run-time architecture Accessed, on 15-4-2009, retrieved from: Aurora: a new model and architecture for data stream management.

Babcock, B., Babu, S., Datar, M., Motwani, R., & Widom, J. (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

Babcock, B., Datar, M., & Motwani, R. (2004). *Load Shedding for Aggregation Queries over Data Streams*. Paper presented at the 20th International Conference on Data Engineering (ICDE 2004). from http://ilpubs.stanford.edu:8090/657/

Balazinska, M., Balakrishnan, H., Salz, J., & Stonebraker, M. (2003). The Medusa Distributed Stream-Processing System: National Science Foundation under Grant No. 0205445.

Balazinska, M., Hwang, J., & Shah, M. Fault-tolerance and high availability in data stream management systems. *Encyclopedia of Database Systems (to appear)*.

Cai, Y., Clutter, D., Pape, G., Han, J., Welge, M., & Auvil, L. (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

Chen, J., DeWitt, D., Tian, F., & Wang, Y. (2000). NiagaraCQ: A scalable continuous query system for internet databases. *ACM SIGMOD Record, 29*(2), 379-390.

Chen, Y., Dong, G., Han, J., Wah, B. W., & Wang, J. (2002). *Multi-dimensional regression analysis of time-series data streams.* In Proceedings Proceedings of the 28th international conference on Very Large Data Bases, Hong Kong, China, 323-334,VLDB Endowment.

Cranor, C., Gao, Y., Johnson, T., Shkapenyuk, V., & Spatscheck, O. (2002). *Gigascope: High performance network monitoring with an SQL interface.* In 623-623,ACM New York, NY, USA.

Daniel J. Abadi, D. C., UˇgurC¸etintemel, Mitch Cherniack, Christian Convey, SangdonLee,, & Michael Stonebraker, N., Stan Zdonik Aurora: A Data Stream Management System, Retrieved from http://mmlab.ceid.upatras.gr/courses/data_mining/docs/par1.pdf

Gedik, B., Andrade, H., Wu, K., Yu, P., & Doo, M. (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., Stonebraker, M., & Zdonik, S. (2003). A comparison of stream-oriented high-availability algorithms. *Brown CS TR-03-17*.

Jain, N., Amini, L., Andrade, H., King, R., Park, Y., Selo, P., & Venkatramani, C. (2006). *Design, implementation, and evaluation of the linear road bnchmark on the stream processing core.* In 431-442,ACM New York, NY, USA.

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.

Linear Road 11/04, 2009, Retrieved from http://www.cs.brandeis.edu/~linearroad/

Madden, S., Shah, M., Hellerstein, J., & Raman, V. (2002). *Continuously adaptive continuous queries over streams.* In 49-60,ACM New York, NY, USA.

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

Nguyen, B., Abiteboul, S., Cobena, G., & Preda, M. (2001). Monitoring XML data on the web. *ACM SIGMOD Record, 30*(2), 437-448.

Palmer, M. Event Stream Processing - A New Physics of Software.(2005), Retrieved from http://www.information-management.com/infodirect/20050729/1033537-1.html

Rizvi, S. (2005). *Complex Event Processing Beyond Active Databases:Streams and Uncertainties*. Accessed on 30-4-2009. Retrieved from http://www.eecs.berkeley.edu/Pubs/TechRpts/2005/EECS-2005-26.pdf

Roberto, S. S. F., Max, S., & David, F. R. (2002). A Web-based Infrastructure for Awareness based on Events.

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

STREAM Data types and operator classes in abstract semantics. , Accessed on 10-5-09, retrieved from http://ilpubs.stanford.edu:8090/641/1/2004-20.pdf.

StreamBase Outbound vs. Inbound Processing, Acessed on : 15-5-2009, retrived from: www.streambase.com.

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

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.

Yao, Y., & Gehrke, J. (2002). The cougar approach to in-network query processing in sensor networks. *ACM SIGMOD Record, 31*(3), 9-18.

Zdonik, S., Jain, N., Mishra, S., Srinivasan, A., Gehrke, J., Widom, J., Balakrishnan, H., Cherniack, M., Cetintemel, U., & Tibbetts, R. (2008). Towards a Streaming SQL Standard. *1*(2), Pages 1379-1390.