UNIVERSITY OF OSLO
Department of informatics

Evaluating the Use of DSMSs and CEP systems in Home Care Domain through a Comparison of TelegraphCQ and Cayuga

Master thesis

Sumandeep Grewal Kaur

19th January 2009

Abstract

Nowadays patients like to have the option of staying at home in their illness period. This can be realized through observing and monitoring the patients motion and temperature with an home care application. Sensors are placed in the home and create a real-time stream which have to be analyzed.

Data Stream Management Systems (DSMSs) and Complex Event Processing systems (CEP) support real-time analysis of data streams which is a continuous ordered sequence of tuples. There are some differences in the design and implementation of these systems. We analyze these differences and similarities for two such systems; TelegraphCQ, a DSMS and Cayuga, a CEP system, with a focus on how well they might suit the needs of home care application domain.

Some of the most important criteria for a system in a home care environment are to have a system with a language that allows an application to formulate appropriate queries. In consideration of this we conclude that compared to Cayuga, TelegraphCQ is the most capable system for home care domain. The reason for this is that the query language of Cayuga has too limited expressiblity. And its operators are too difficult to use for expressing the queries that are typical in this application domain.

TelegraphCQ in contrast does report the correct value and time interval with the possibility to use built in functions from SQL and do not demand query with lot of complexity.

The comparison is done in two parts; theoretical and practical. We focus on topics like tuple definition, aggregation, consecutiveness, concurrency and optimization. Through the first part we investigate the literature concerning the two types of systems. The second part consist of testing these systems in the home care application domain. This part allows us to test the topics discussed in the theoretical part.

Preface

This Master's Thesis is written at the Distributed Multimedia Systems Research Group at the Department of Informatics, University of Oslo, during the semester of autumn 2008.

I want to thank my supervisors, Ellen Munthe-Kaas and Jarle Søberg for their guidance, vital feedback and support. This have been of great importance to me. I would also like to thank Hashmat Khan to have read through the thesis and given me great feedback.

Sumandeep Grewal Kaur
University of Oslo
Januar, 2009

.

Contents

FIGURE LIST

EXAMPLE LIST

1

INTRODUCTION

In the home care application domain one needs to monitor different events generated by preplaced sensors nodes. One example is to monitor a patient which has been sent home. Having patients in their homes instead of in hospitals or nursing homes is important. It gives the patient a familiar environment and the hospital resources to treat more severe injuries or illness. For instance, long durations of non movement can indicate a problem and that a rescue unit needs to be sent to the home for checking the patient.

By automating the home care application domain, several questions arise with respect to which type of system to use:

- How important is it that the system reports the exact value?
- How important is it that the system reports the right time interval?
- How important is it to have a reliable system?

Our opinion is that the home care domain is a domain which can be described as a sensitive domain. With a system in such a domain it is important that the system is able:

- to catch a problem fast and alert the rescue unit.
- to report the exact value
- to report the right time interval
- to manage large amount of data
- to have a reliable and stable system
- to maintain the privacy of the patient (not taken into consideration).

Two technical possibilities which can be applicable for the home care domain is Data Stream Management Systems (DSMSs) and Complex Event Processing systems (CEP).

DSMSs and CEP support real-time analysis of data streams, which is a continuous ordered sequence of tuples. There are some differences in the design and implementation of these systems. We analyze these differences and similarities with focus on home care domain by comparing TelegraphCQ, a DSMS with Cayuga, a CEP.

TelegraphCQ is an adaptive system that should be able to repeatedly measure a home care environment. Using a dataflow engine TelegraphCQ is able to move large amounts of data through an amount of operators running on one or many computers. TelegraphCQ uses new dataflow technologies to route unpredictable

and fractured dataflow through computing recourses on a network, resulting in manageable streams of useful information.

Cayuga is a system that can be used in the home care domain since it is a complex event monitoring system for high speed data streams, which supports on-line detection of a large number of complex patterns in event streams. Based on nondeterministic limited state automata with buffers, Cayuga manages to merge a simple query language for composing stateful queries with a scalable query processing engine. It is not only able to scale with the arrival rate of events in the stream, but also with the number of queries, which is an important feature.

In addition to do research on the technology we are also interested in discovering differences and similarities between Cayuga and TelegraphCQ based on already available information and research work. To underline these differences and similarities we focus on vital and relevant topics. Through comparing these two systems we also discover if they are capable of being used as a technical solution in a home care environment, where the outcome is crucial.

The comparison is done in two parts; theoretical and practical. The theoretical part focuses on topics like:

- How *tuple and event* are defined in each system.
- We investigate the five standard types of *aggregates*; AVG, SUM, MIN, MAX, and COUNT.
- We investigate how well the systems manage to capture *consecutive* events.
- If the systems can query and *join* the information of results from several data sources and *concurrent* execution of several independent queries.
- To what extent is *optimization* possible in the systems.

The practical part is based on the theoretical part. In this part we go through five relevant tasks and analyze and evaluate how well the systems in practice can be used in a home care environment. The goal is see how the two systems solve the same type of queries on the same event stream.

The rest of this thesis is organized as follows. We begin with Chapter 2 by covering the background of DSMSs and CEP. This chapter also consists of general information about Cayuga and TelegraphCQ together with their query language. In chapter 3 we focus on the topics; tuple/event definition, aggregation, consecutiveness, concurrency and optimization, based on already existing research work and information. Further in chapter 4 we test the topics on each of the systems. The last chapter consists of conclusion and further work.

2

BACKGROUND

This chapter is an introduction to Data Stream Management Systems (DSMSs) and Complex Event Processing system (CEP). Further we continue with general information about Cayuga and TelegraphCQ together with their query languages and some application domain.

2.1 DSMS in general

The content in this chapter is mainly based on [DSS], [DSM] and [ICI].

A data stream is a sequence of tuples. Tuples are streaming, not stored in a table. Each tuple consists of a set of attributes, similar to a row in a database table. Data Stream Management Systems (DSMSs) support on-line analysis of (such rapidly changing) data streams. Different from traditional Database Management Systems (DBMSs) where data is stored on disks and the queries are performed against the stored data, in DSMSs there is no storage of data on disk, all operations are performed in main memory, see Figure 1 [ICI].

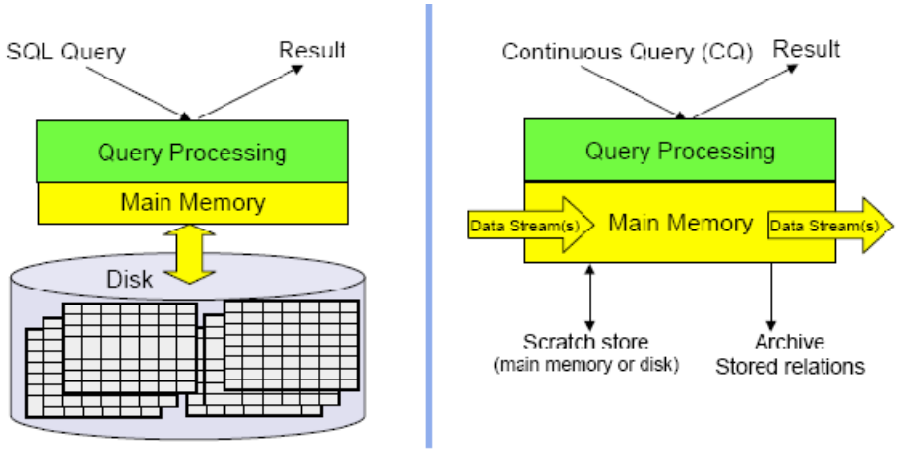


Figure 1: DBMS vs. DSMS [INF5090]

In DSMSs the information is arriving in main memory as an unpredictable stream of data. The data stream is unbounded and ordered; implicitly by arrival time or explicitly by timestamp. An input monitor regulates the input possibly by dropping packets or transactions as well as preprocessing and buffering. The data stream is too large to be stored entirely in main memory. For that reason data is stored in

three partitions in main memory: the *Working Storage*, e.g. for window queries, the *Summary Storage* for stream synopses, and the *Static Storage* for meta-data, e.g. physical location of each source. In addition a *Query Repository* is used for long-running, continuous queries. A *Query Processor* communicates with the *Input Monitor*; it may also re-optimize the query plans in response to changing the input rates on the data. Results are then temporarily buffered through an *Output Buffer* and streamed to the user and, see Figure 2 [DSM].

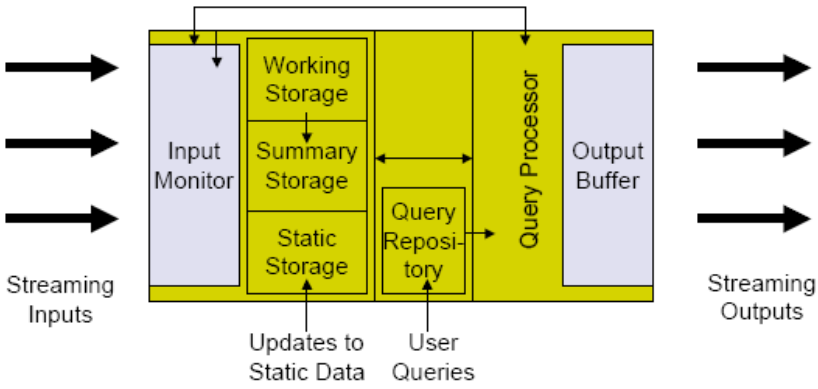


Figure 2: Generic DSMS Architecture [INF5090]

To control the order in which items arrive and locally store a stream in its entirety, is impossible. Queries over streams run continuously over a period of time and incrementally return new results as new data arrive. These are known as *long-running, continuous, standing*, and *persistent* queries. Requirements to a DMMS are formed by the characteristics of data streams and continuous queries: (1) order-based and time-based operations must be allowed by the data model and query semantics, e.g. queries over a five-minute moving window, (2) the use of approximate summary structures, in the literature referred to as synopses or digests; as an outcome, queries over the summaries may not return precise answers, (3) not using blocking operators which consume the entire input before any results are produced, (4) not possible for backtracking over a data stream due to performance and storage constraints; on-line stream algorithms are restricted to making only one pass over the data, (5) applications that monitor streams in real-time must react quickly to unusual data values, (6) long-running queries may run into changes in system conditions throughout their execution lifetimes for instance, variable stream rates, and (7) to ensure scalability, shared execution of many continuous queries are needed.

A method for managing such data when the queries will go over a period is to use reduction techniques, which imply *sampling, shedding,* and *synopsis*. This implies to respectively only use a part of the data which has arrived, reduce or drop parts of the data, or summing up data in several ways. There are three main ways of managing data which has arrived: *accuracy*, *history* and *real-time*.

Accuracy; because of the limited memory it is not possible to collect the accurate answer, but instead it is possible to obtain summary or headword.

History; for looking at the most recent data, windowing has to be deployed. There are three types of windows: *sliding windows* which is used if it is important to gather all the information, *jumping windows* is advantageous to use and as result sampling will be received, and *overlap windows* which are known as the least effective windows; it uses the same data several times but with a lot of queries it can be reasonable to deploy it.

The last way of managing data is *real-time* which handles the data immediately and therefore cannot accommodate everything in advance. For this reason the query plan is made at the same time, here and now or on the way in contrast to conditions where the query plan already exists.

To conclude this section about DSMSs we will portray basic continuous query operations over streaming data: (1) *selection*; all streaming applications require support for complex filtering, (2) *nested aggregation* are needed to compute trends in the data, (3) *multiplexing and demultiplexing* are used to decompose and merge logical streams, (4) *frequent item queries*, also known as top-k or threshold queries, are dependent on the cutoff condition, (5) *stream mining* is for on-line mining of streaming data; operations like pattern matching, similarity searching and forecasting are needed, (6) *joins* are used to support multistream joins and join of streams with static metadata, and (7) *windowed* queries; all of the query types mentioned above may be constrained to return results inside a window.

2.2 CEP in general

In the mid 1990's academic research in automated analysis of event traces formed the beginning of Complex Event Processing (CEP) at Cal Tech (Mani Chandy), Cambridge University (John Bates), and Stanford University (David Luckham). It was now focus on giving more detailed information about the behavior of the models into the event traces, like the cause and effect between events, by developing models to process streaming event data by identifying complex sequences events with temporal and spatial constraints, and to control complex actions as a outcome of these patterns [CWW] [CAGP].

A complex event is when no one can directly detect the situation; one has to conclude or assume that the situation has taken place from a combination of other events. CEP helps detect such complex, inferred events by analyzing and correlating other events. It is a technology that can aggregate, analyze and respond immediately to real-time event data, with minimal latency [CHPE] [TEPS].

In more detail CEP is a technology for extracting information which can be low level network processing data or high level enterprise management intelligence, depending on the users' interest. It is detecting pattern or trends that characterize incidents or warnings in real time, as they happen so one can respond immediately. In CEP you can know your position at all time by combining data

from multiple sources and continuously computing aggregate values. CEP gives you also the possibility to adjust the changing conditions by constantly observing the interaction of data [TEPS] [CEP].

2.3 Cayuga

This chapter is based on content from [CHPE] [CAGP].

Cayuga is described as a complex event monitoring system for high speed data streams, which supports on-line detection of a large number of complex patterns in event streams.
Based on nondeterministic limited state automata with buffers, Cayuga manages to merge a simple query language for composing stateful queries with a scalable query processing engine. It is not only able to scale with the arrival rate of events in the stream, but also with the number of queries which is an important feature.

The Complex patterns in event streams which is mentioned above are described by using a query language based on composable operators which have well-defined formal semantics. With this Cayuga is able to perform query-rewrite optimizations and can build up complex patterns from simpler sub-patterns.

The Cayuga system also implements several techniques for query processing, indexing, and garbage collection, resulting in an efficient execution engine that can process data streams at very high rates.

Cayuga also supports resubscription, the output event stream from one query can be used as the input stream to one or more other queries. Resubscription enables complex event pattern queries, and it extends the expressiveness of the query language.

Cayuga has a web-based frontend which is running on a custom Python Web server, with AJAX-based controls for asynchronous communication and user-friendly interfaces in the browser. The users can enter persistent queries and register them with a running Cayuga engine. Users are given a choice of predefined templates from a dropdown menu. Then, they can modify these to their need, or write one from scratch.

2.3.1 Cayuga query language

The Cayuga query language(CEL) is a result from an event algebra. It is a mapping of the operators in algebra into a SQL like syntax. Each query has the following form:

SELECT ⟨*attributes*⟩
FROM ⟨*algebra_expression*⟩
PUBLISH ⟨*output_stream*⟩

Example 1: Cayuga Query Form[CHPE]

The SELECT section state the attributes in the output stream schema, the FROM section indentify a Cayuga event pattern, and the PUBLISH clause gives the output stream
a name.

The event pattern can be built with three different operators; FILTER; NEXT and FOLD. The FILTER { θ } operator selects those events from the stream that satisfy the predicate θ. The second operator NEXT { θ } allows to correlate events over time. When applied in a query it combines each event from two streams which satisfies the predicate θ and occurs after the detection time of the event in the first stream. The last operator FOLD { θ } is a generalization of the NEXT. The difference is that FOLD looks for patterns comprising two or more events. It is used in situations where we need to iterate over an a-priori unknown number of events until a stopping condition is satisfied [CAGP].

2.4 TelegraphCQ in general

This chapter is based on content from [TWW].

TelegraphCQ is an adaptive dataflow system . Using a dataflow engine TelegraphCQ is able to move large amounts of data through an amount of operators running on one or many machines. Dataflow operators can be compare d with loops, which repeatedly receive data from their inputs, and place data on their output. All the operators used in Telegraph are pipelining, which means that they produce data to their output before they finish receiving all the data from their inputs. Thus TelegraphCQ uses new dataflow technologies to route unpredictable and fractured dataflow through computing recourses on a network, resulting in manageable streams of useful information.

As an adaptive system TelegraphCQ should be able to repeatedly measure its environment, and decide how to take actions based on such measurements. The design of TelegraphCQ is predicated on the assumption that most of the new computing problems will take place in very unpredictable environments. TelegraphCQ consist of two basic adaptive dataflow techniques, Eddies and Rivers. Eddies are used to continuously remodel dataflow graphs to maximize performance. Eddies provide to adaptively route data through operators, continuously changing the order of operators in a dataflow graph by observing the rates at which operators consume and produce data. An eddy is implemented as a special iterator that can be injected into any dataflow graph to make it adapt its shape. Rivers are used to continuously load-balance work across multiple machines on a network.

Rivers adaptively route data to different machines, by observing the rates at which the machines can handle additional work.

TelegraphCQ is implemented as an open-source DSMS prototype, based on the PostgreSQL database system.

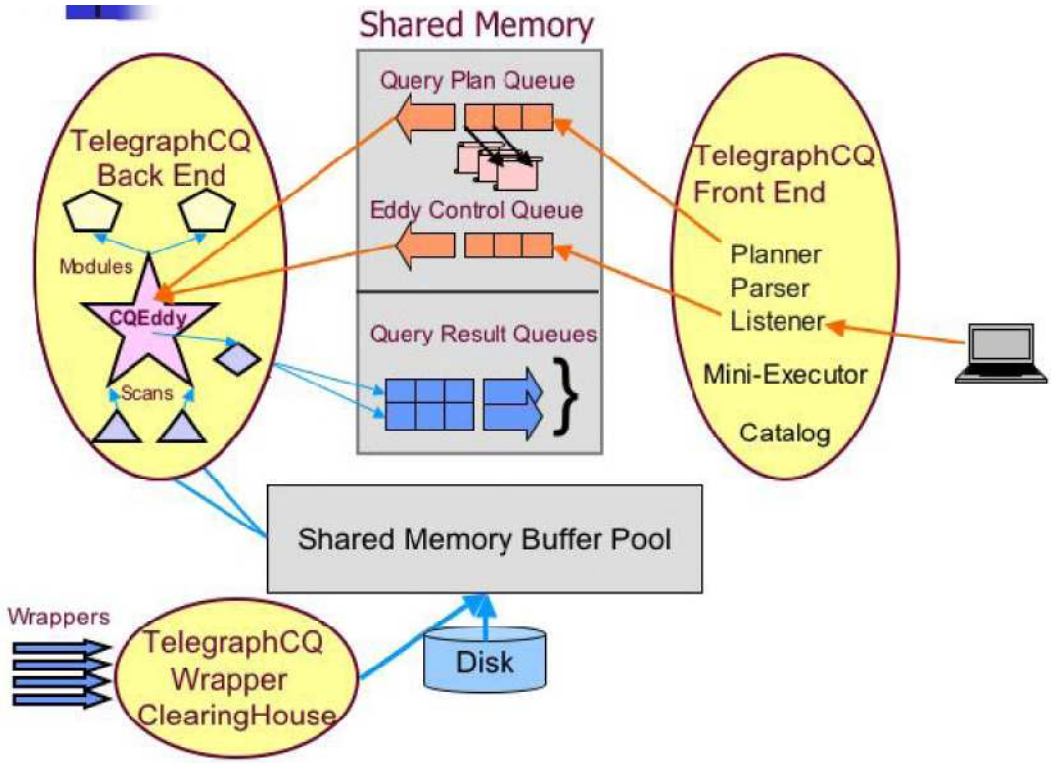


Figure 3: The TelegraphCQ Architecture [TWW]

In the TelegraphCQ architecture in Figure 3 we can see that a separate PostgreSQL process TelegraphCQ Front End (TFE) is forked for each client connection. This process runs all queries that do not involve streams using the normal PostgreSQL executor. When PostgreSQL receives a continuous query, the TFE parses and plans the query in shared memory. It uses the output of the PostgreSQL optimizer to construct a continuous query plan.

Further, the TFE passes the plan to the TelegraphCQ Back End executor (TBE) using a shared memory queue. The TBE runs the TelegraphCQ eddy which merges all continuous query plans into one so that query processing may be shared amongst queries. The TBE receives the plan, and integrates it into the TelegraphCQ eddy. Finally, the Eddy returns results to the appropriate TFE via shared memory result queues, one per query.

2.4.1 The Query Language of TelegraphCQ

The query language of TelegraphCQ is also comparable with SQL. It is a straightforward extension to SQL for manipulating streams. Each query has the following form:

```
SELECT <select_list>
FROM < stream_list> < interval-expr>
WHERE <predicate>
GROUP BY <group_by_expressions>
ORDER BY <order_by_expressions>;
```

Example 2: TelegraphCQ Query Form [TWW]

The SELECT section state the attributes in the output stream schema, the FROM section state the stream with a interval expression. WHERE state the conditions. GROUP BY and ORDER BY state the ordering.

The interval expression in FROM clause is the window section. Using the window section we can specify streams with sliding, tumbling or jumping time window. Parameters used for these in the query language is respectively ”RANGE BY…”, “SLIDE BY…” and “START AT…”.

The parameters have different functional purpose, the RANGE parameter in a query defines the size of the window in a specified time, as an example we can state the window section in the query accordingly: RANGE BY ´5 minutes´. With the SLIDE parameter we can define the interval after which this window will be re-calculated, again by specifying a time, as an example to this: SLIDE BY ´1 minutes´. And the START parameter define the time at which the window begins, an example on this will be: START AT ´2008-09-09 12:12:00´. All the three parameters above can be stated separately or in the same query [TWW].

2.5 Existing applications for DSMSs and CEP systems

Efficient asynchronous interaction among distributed applications has been an active field of research for many years now, with focus on topics like spanning active databases, event systems, high performance implementations of Publish/Subscribe, and distributed Publish/Subscribe [CWW].

In complex event processing users are interested in finding matches to event patterns which are usually sequences of correlated events. A case of such a pattern is a safety condition; in safety condition the users want to make sure that nothing bad happens between two events which are set by condition and terms which corresponds users requirements. An example of this kind of pattern: ”between leaving the farm (start event) and arriving at the store (end event), fresh

produce should not have spent more than 1 hour total above a temperature of 25℃"[CAGP].

2.5.1 Sensor Networks

In today's world it is possible to monitor physical or environmental conditions with a sensor network. For instance observing and analyzing human behavior or detection of forest fire are applications where wireless sensor networks can be helpful.

The motivation for the development of sensor networks was originally military applications such as battlefield surveillance. The aim was monitoring friendly forces, equipment and ammunition, reconnaissance of opposing forces and terrain, and nuclear, biological and chemical attack detection and reconnaissance. Now sensor networks are used in many other applications as well; civilian application areas include environment, health care applications, home automation, and traffic control.

The target of environmental applications is to track movements of birds, animals, etc. and to monitor environmental conditions that affect crops and livestock. It can also detect chemical/biological, monitor earth and environmental in marine, soil, and atmospheric contexts. Environmental applications also helps meteorological or geophysical research, pollution study, precision agriculture, irrigation. Or biocomplexity mapping of environment, detection of forest fire or flood[INF5090].

Health application applies integrated patient monitoring and telemonitoring of human physiological data. Can also be used for tracking and monitoring doctors and patients inside a hospital. Or tracking and monitoring patients and rescue operations[INF5090].

Commercial applications are for instance used for monitoring product quality or machine diagnostics. It can also provide work for construct smart office spaces or smart structures with sensor nodes embedded inside. Interactive toys or museums. Managing inventory control. Environmental control of office buildings. Commercial applications can also be used for detecting and monitoring car thefts or for vehicle tracking and detection[INF5090].

A sensor network is a collection of autonomous sensor nodes which consist of sensing, data processing, and communicating components [WSN01]. Every sensor node is equipped with a *sensing* unit; usually composed of sensors and analog to digital converters (ADCs), *processing* unit; manages the procedures that make the sensor node collaborate with the other nodes to carry out the assigned sensing tasks, *transceiver* unit; connects the node to the network or satellite, and a *power* unit; usually a battery and the most important unit [INF5090]. The sensor node receives waves or other variations from one system and transmits related ones to another with a communications infrastructure aim to monitor and collect physical data. Based on sensed physical effects and incidents the processing unit manages the procedures that make the sensor node

collaborate with other nodes to carry out the assigned sensing tasks. Transceiver unit receives commands from a central computer (base station), connects the node to the network and transmits data [INF5090].

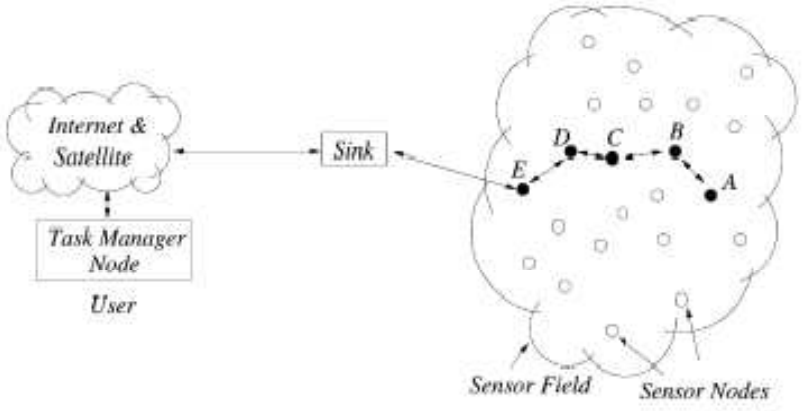


Figure 4: Sensor nodes scattered in a sensor field [WSN01]

In the figure above [WSN01] we can see sensor nodes spread in a sensor field, they are deployed either inside the incident or very close to it. These sensor nodes duties are to collect data and route data back to the sink and the end user. Through the sink in the figure, data are routed back to the end user by a multihop infrastructure less architecture. The sink may communicate with the task manager node via internet or satellite.

Wireless sensor networks also have some limitations, the most important relates to; *communication, power consumption, computation and uncertainty in sensor readings*.

- *Communication*: The wireless network connecting the sensor nodes constraints to limited quality of service, latency with high variance, limited bandwidth, and frequently drops packets[QPSN].
- *Power consumption*: Energy conservation is an important system design considerations of any sensor network application, because sensor nodes have limited supply of energy. An example of this is MICA mote from Berkeley, the motes is powered by two AA batteries which provide about 2000mAh, powering the mote for approximately one year in the idle state and for one week under full load [QPSN].
- *Computation*: Sensor nodes have restricted computing power and memory sizes. This limits the types of data processing algorithms on a sensor node, and it limits the sizes of intermediate results that can be stored on the sensor nodes [QPSN].
- *Uncertainty in sensor readings*: Sensor malfunction might generate inaccurate data, and unfortunate sensor placement such as a temperature sensor directly next to the air conditioner might bias individual readings [QPSN].

2.5.2 Network Monitoring

Large networks are growing complex by increasing demands, over provisioning, hardware changes, and manual configuration, etc and are consequently difficult to manage. Therefore it is necessary to monitor and analyze the network traffic flowing through the systems [GIGA].

Network monitoring implies the use of systems that constantly monitors a computer network for slow or failing systems and then alerts the network administrator in case of outage through alarms. A network monitoring system detect problems caused by overloaded or crashed servers, network connections or other devices [Wiki]. But the monitoring requirements vary from the long term such as monitoring link utilization, computing traffic matrices to the ad-hoc such as detecting network intrusions, debugging performance problems [GIGA].

As the internet continues to grow fast both in size and complexity, it has become gradually more important to have tools to analyze the internet traffic. Many of the tools which exists are complex (e.g., reconstruct TCP/IP sessions), query layer-7 data (find streaming media connections), operate over huge volumes of data (Gigabit and higher speed links), and have real-time reporting requirements (e.g., to raise performance or intrusion alerts) [GIGA].

As mentioned before the need of analyzing internet traffic is increasing and many tools for analyzing internet traffic exist, such as ISP monitor service levels, identify bottlenecks and so on. Further we have an example of traffic analysis which analyze internet traffic in near real-time to computer traffic statistics and detect critical conditions. Figure below describes the basic structure of tools for network monitoring which consist of packet capturing, trace file, analysis and in the end result [INF5090].

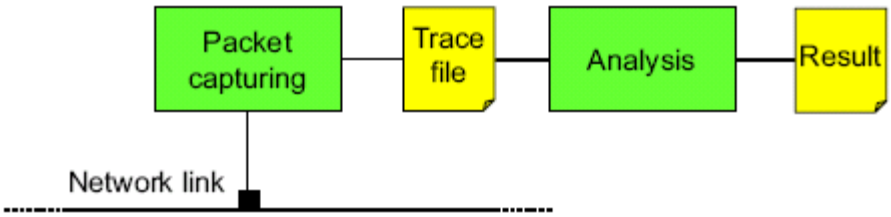


Figure 5: Basic structure of tools [INF5090]

With this basic network monitoring you are able to analyze data both online and offline. Packet capturing happens in real time hence online monitoring is possible. With trace file it is possible to monitor offline because the data is stored. The same tool is used to both manage and analyze data (trace file and result).

2.5.3 Stock Trading

An example on stock trading is stock ticker event monitoring, a system that allows financial analysts to compose subscriptions over a stream of stock ticks. It is about on-line analysis of stock prices, discover correlations and identify trends, it is about predicting future price. In this field technical analysts look for many different patterns in price movements. The interpretations of such patterns are used to support trading decisions. Investors that are able to see trends before other market players will be able to make early moves and thus increase profit margins.

A well-known pattern amongst analysts is the double top pattern; the analysts are interested in being notified whenever there is double top formation in the price chart of any stock[CWW].

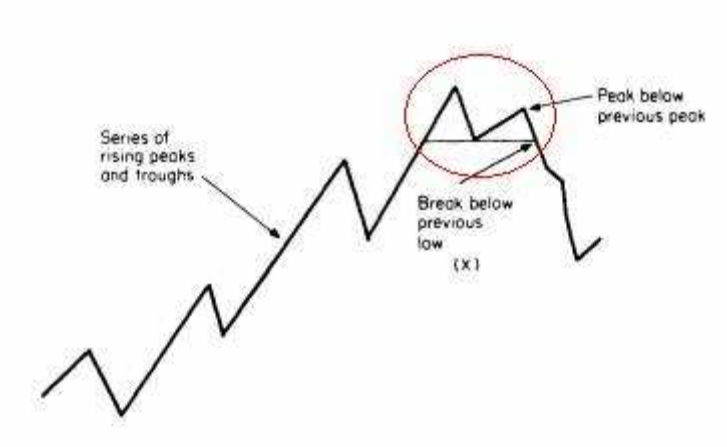


Figure 6: Double top formation marked in red [CWW]

2.6 Home Care Application

In situations were people are sick, disabled or old there might be a need for nursing and observation of these. But due to different circumstances it is not always possible or desirable to be taken care of. Finances can be one of the reasons where the hospital for instance can not afford the expenses, or they do not have the facilities like nurses, beds, etc. As mentioned it is not always desirable to be taken care of, the reason for this can be that people for instance want to live a normal life as possible.

These people in different situations can still be taken care of with the home care application. With home care application a person can live a normal life without being dependent on others. Hospitals which do not have the facilities like nurses can observe patients with such home care applications.

The home care application domain covers the need for monitoring different events generated by preplaced sensor nodes. As an example motion sensors

can be placed around the home to monitor the movement of a patient. Long durations of non movement can then indicate a problem and that a rescue unit needs to be sent to the home.

Disabled persons can manage home devices easily and remotely by for instance with embed sensor nodes and actuator in different units such as vacuum cleaners, TVs and refrigerators. Such sensor nodes inside the unit can interact with the external network via the Internet or Satellite [UCEP].

2.7 Cayuga and Application Domain

Cayuga is like traditional Publish/Subscribe system but predicate filtering of events, correlation of events and aggregation of events. Standard Publish/Subscribe is a dominant paradigm where users are allowed to express stateless subscriptions; these are evaluated over each event that arrives at the system [CWW].

Cayuga is designed for pattern matching queries. Cayuga on a single PC can scale up to hundreds of thousands of concurrently running pattern matching queries. So the main distinguishing feature of Cayuga is its scalability. But it exist queries which are not expressible in Cayuga. The aim of Cayuga is occupy a spot between simple Publish/Subscribe and the full power of SQL-like queries in DSMSs, and achieves greater expressiveness than Publish/Subscribe while keeping most of its advantages in terms of scalability[CWW].

2.8 TelegraphCQ and Application Domain

TelegraphCQ contain two types of application: pull-based and push-based applications [TCDP]. As in traditional database systems the pull-based applications pull data from the disk to the query processor by the user upon demand. And in push-based application data are constantly “pushed” from disk to the query processor out to the user.

An example on pull-based application is sensor network. In sensor network the sensors pull data from the environment depending on the sensing device, and as shown in the chapter about Sensor Networks above the sensors send data through the network back to a central node for querying and data analysis.

Network monitoring, Stock trading and Home Application is examples on push based application. Applications like these produce streams which are intense loaded with data. And as a result of this these systems cannot control the arrival frequency at which data elements are received.

3

Comparison of Cayuga and TelegraphCQ

We are interested in discovering differences and similarities between Cayuga and TelegraphCQ based on already available information and research work. To underline these differences and similarities we focus on the following five vital and relevant topics; 1) Tuple/event definition, 2) Aggregation, 3) Consecutiveness, 4) Concurrency and 5) Optimization.

3.1 Tuple/event Definition

Generally speaking an event is described as an occurrence indicating that something has happened or is in the process of happening. As a practical example an event can be a sensor reporting the movement of an object, it can be a change in the financial market or it can be a seismic sensor sensing an earthquake.

Event is represented in the system as a tuple, a sequence of values. Each value is called a component of the tuple. These tuples form an event stream. An event stream is apossibly unbounded set of timestamped tuples. The timestamp can be externally provided or by the system.

Event stream in Cayuga has fixed relational schema, and events in the stream are treated as relational tuples. Each event has two timestamps, a start time and a detection time, also called end time. Events can have a non-zero but finite duration. The detection times determine the order of events. Event with the same detection time are considered to happen simultaneously. Cayuga processes events in epochs. Through one epoch all events with the same detection time are processed [General].

In Cayuga set of tuples in event stream occurs in this arrangement <ā; t_0, t_1 >. Here $\bar{a} = (a_1, \ldots . a_{n)}$ are data values for the corresponding attributes and t_0 , t_1 are values representing the start and end timestamps of the current event. Tuples in Cayuga is timestamped by the system. Below is an example on a tuple [TEPS].

"Assume the stream of stock sales published by the data source has fields (name, price, vol; timestamp). An event from that stream then could be the tuple <IBM, 90, 15000;9:10,9:10>"

Example 3: Cayuga tuple example [TEPS]

It is various type of events [CEP]; for instance simple -, complex -, raw -, derived – and instantaneous events. A simple event does not represent other events, it is not an abstraction of other events and it is neither composition of other events. While a complex event is an abstraction of other events called its members which can be simple events or other complex events. Raw events may represent both simple event and complex event; it is an event object that records a real-world event. An event that is generated as a result of applying a methods or process to one or more other events is called derived event or synthesized event. For instance the absence of an event in a given time interval can lead to a derived event reporting that the first event did not happen. An instantaneous event object will have a single timestamp signifying when the event happened. This applies where the start and end times are the same.

When it comes to TelegraphCQ a stream is an infinite bag of events < s, t > pairs. Where *s* is the tuple and *t* is the timestamp of the event [CQL]. Timestamp specifies the creation time of a tuple. As opposed to Cayuga tuples can be externally timestamped, or timestamped by the system. If they are time-stamped by system, they will be in monotonically increasing order [TWW].

We have two types of data streams in TelegraphCQ; archived and unachieved. The tuples that are streamed into archived streams will be copied onto disk by TelegraphCQ. Tuples that are streamed into unachieved streams are discarded when the query is cancelled [TWW].

Below is an example on tuple in TelegraphCQ shown:

Assume an unarchived stream of stock sales published by data source has fields (name, price, vol; timestamp). An event from that stream then could be the tuple <IBM, 90, 15000, 9:10>

Example 4: TelegraphCQ tuple example

If we compare this example from TelegraphCQ with the example above from Cayuga we will for instance notice the difference between the timestamps. As we have mentioned before In Cayuga the timestamp is defined with a start time and a detection time (also called end time), while in TelegraphCQ the timestamp only specifies the creation time of the tuple.

3.2 Aggregations

An aggregate function is a function that returns a single value from a collection of input values such as a set, a bag or a list. As described earlier streams are unbounded. This makes aggregate queries difficult to process, since they cannot produce any results unless they have observed all their input. A way to handle this is to limit the number of input tuples to aggregate query over streams. But let us see how Cayuga and TelegraphCQ manage aggregation.

Cayuga compare its query language CEL with SQL, but SQL has a lot of built-in functions for counting and calculations which CEL doesn't have. For instance SQL has the five standard types of aggregates such as; AVG, SUM, MIN, MAX and COUNT. CEL must instead take help of 'and', 'as', =, <=, <, > and >= to write query from.

Cayuga's algebra contains three aggregate functions. When the topic about aggregation is discussed in the sources [e.g. TEPS, CWW] it is stated that in Cayuga's algebra aggregate functions fit naturally in. The aggregation occurs over a sequence of events. But in further research there is no trace of the five standard types of aggregation in Cayuga's query language. Queries where a standard aggregate function is needed has been solved in different ways using Cayuga's built-in functions. Below we have shown examples of how the aggregate functions can be implemented.

"Notify me when the difference between the current price of a stock and its 10 day moving average is greater than some threshold value."

```
SELECT Name, sum / count AS Price
FROM FILTER{DUR = 10day} (
        SELECT *, Price as sum, 1 as count FROM S
        FOLD {Name=$1.Name, , $.count + 1 as count, $.sum + Price as sum} S)
PUBLISH MovingAverage

SELECT *
FROM MovingAverage NEXT {Name = $1.Name, Price - $1.Price > threshold} S
```

Example 5: Cayuga Average example [CWW]

Above is an example where they could have used aggregate functions like AVG and SUM. In this example it has also been used divide and conquer to express the query more easily. Resubscription is used to allow query to subscribe to the output of another, so the PUBLISH clause can specify an identifier of the output stream which other queries can specify as their input stream. This is possible because the operators in Cayuga are stream-to-stream, assuming that the input and output of every operator must be a stream. This example also show that

queries can be nested by declaring the output from an inner SELECT clause as the input for the FILTER operator (see line two) [CWW].

Further an example for COUNT:

```
Notify me when a camera with some feature (like 12X zoom) is released by Canon
and there are at least 10 positive reviews on this product within a given time span.

SELECT *
FROM FILTER {DUR <= w and count >= 10}
(FILTER{Company = 'Canon' and Category = 'Camera' and CONTAINS(Description,
'12X zoom')}
(SELECT *, 1 as count FROM Products) FOLD {CONTAINS(Content, $1.URL) and
IsPositive(Content), , $.count + 1 as count} S)
```

Example 6: Cayuga Count example [CWW]

An example for MIN and MAX:

```
Notify me when the lowest price in a (t, t+5 minute) window is greater than the highest
price in a (t',t'+5 minute) window, where t'>t and t'-t is less than some constant value.

Let t' - t <= threshold

SELECT Name, MinPrice
FROM FILTER{DUR = 5min} (
        SELECT *, Price as MinPrice FROM S
        FOLD {Name=$1.Name, , min($.MinPrice, Price) as MinPrice} S)
PUBLISH T_WINDOW

SELECT Name, MaxPrice
FROM FILTER{DUR = 5min} (
        SELECT *, Price as MaxPrice FROM S
        FOLD {Name=$1.Name, , max($.MaxPrice, Price) as MaxPrice} S)
PUBLISH T_PRIME_WINDOW

SELECT *
FROM FILTER {DUR <= threshold} (
T_WINDOW NEXT {Name = $1.Name, MaxPrice < $1.MinPrice} T_PRIME_WINDOW)
```

Example 7: Cayuga MIN and MAX example [CWW]

Other aggregate functions in Cayuga for instance like GROUP BY is also solved in their algebra [TEPS], but no trace of it in CEL examples.

To handle a limited portion of the stream for aggregate queries, TelegraphCQ apply a window section to it's query form to create a time-varying relation. This

allows us to retrieve results without waiting for the entire data stream to be processed.

This windowing technique supports both the portion of streams that has already arrived, as well as those portions that will arrive in the future. As mentioned earlier TelegraphCQ has two kind of applications; pull-based and push-based applications, execution of the query over consecutive windows is possible on both [TCDP].

Windowing is needed if we want to use aggregation in our queries, since we need to bound the aggregate state. Hence aggregates on streams are computed on windows. Below is an example on aggregation [TWW]:

```
SELECT
  SUM(R.i), AVG(R.j), COUNT(*), wtime(*)
FROM
  R [RANGE BY 't1 seconds' SLIDE BY 't2 seconds'];
```

Example 8: TelegraphCQ Aggregation example [TWW]

When the query execute an aggregate, for instance grouped, it is based on the interval expression of the different streams in the query.

TelegraphCQ has solved aggregation query like SQL with the aggregate functions, but with help of windowing. Cayuga has solved the aggregation in its algebra, though it convey the impression of that their query language is also very alike SQL the aggregation is missing in CEL. It is possible to solve query where aggregation could have been used in Cayuga, but as we can see in the given examples it requires more.

As TelegraphCQ, Cayuga is also using a form of Windowing but it is different. TelegraphCQ use time-based windowing while in Cayuga we have to make the window with help of PUBLISH function before we can use aggregation. The window of Cayuga can this way contain more. But after publishing a window in Cayuga you can still not use the five standard aggregate functions, neither does Cayuga has any other function which can replace these.

3.3 Consecutiveness

The concept consecutiveness is generally explained as periods of time or events happen one after the other without interruption. If we look at det figure below event B is a successor to event A because no event happened in between of these two events. Hence is event C not a successor of event A because event B happened in between them.

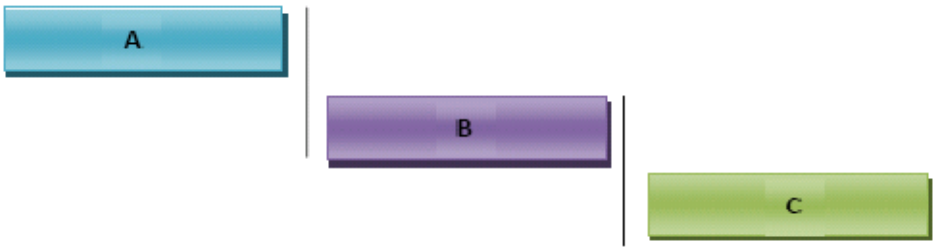


Figure 7: Consecutiveness with linearly timestamp

But as we know Cayuga and TelegraphCQ use different timestamp for managing their data streams. (Cayuga use start and detection time, while TelegraphCQ use only creation time). This can also cause to different understanding of consecutiveness for each of them. As we know event systems are used to analyze time series queries in real time. It is therefore an important factor that these systems have some semantics for consecutiveness; when is one event a successor of another.

Let us begin with the consecutiveness for Cayuga first. Cayuga is a system where the outputs of a query are themselves events, which can be posted to the event stream and used in other queries. Better known as complex events, since they contains of several smaller events that together satisfy the query. Also the query mentioned above might be a complex event since it contains several steps [NEXT]. Therefore Cayuga use interval timestamps because these events may have duration. For this reason complex events may overlap with each other. This is something which can cause some difficulty in processing the sequencing operator because for events with duration the successor is not obviously to state.

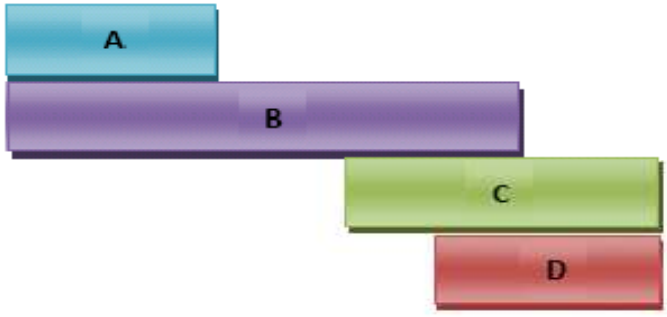


Figure 8: Consecutiveness with interval timestamp

Figure above is an example on interval timestamp. If we have to choose the successor according to the end time of the interval, then event B is the successor of event A. But if we look at event B it properly contains event A, so it may be a reason to exclude it as a successor of event A, we do not want to "skip" over some data. So instead should either event C or event D be the successor of event A or both of them.

As mentioned before in CEP systems users register long running queries to detect event patterns, which are typically sequences of events. An example on such query can be to detect when something bad happens between to events;

Post a notification if an item, after being removed from shelf, exits the store before being checked out at the counter [NEXT].

To solve this type of queries all event systems has sequencing operator, also described as a concatenation operator, E1; E2. This operator finds any event matching the sub pattern E1, and then finds the first match afterwards to the sub pattern E2. But from previous finding Cayuga does not have this operator in it's query language. Question which is still unanswered is if it only finds E2 which follow E1 or if it also look at if E2 follow immediately after E1[NEXT].

As mentioned Cayuga uses interval time stamps, where $t = [t_0, t_1]$ is a successor of $s = [s_0, s_1]$ if $t_0 > s_1$ and there is no event with time stamp $p = [p_0, p_1]$ such that $s_1 < p_0 < p_1 < t_1$. That is, t is a successor of s if t follows s without overlap, and no p that follows s without overlap finishes before t. This avoids unbounded successor sets with their associated implementation difficulties. Explained in the figures below.

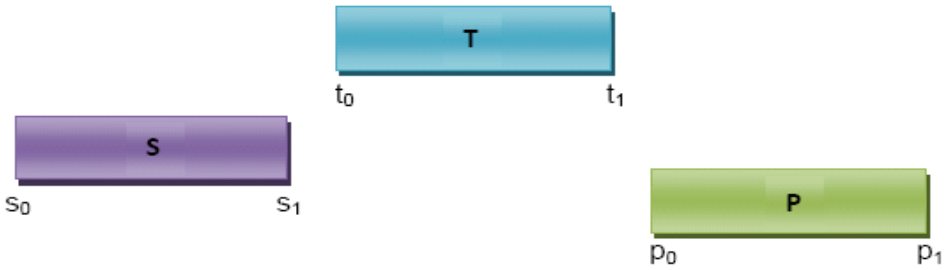


Figure 9: Cayuga event T successor of event S

In this figure event T is successor of event S because as we can see $t_0 > s_1$ and that event P does not appear in between of them. $s_1 < p_0 < p_1 < t_1$ is not correct.

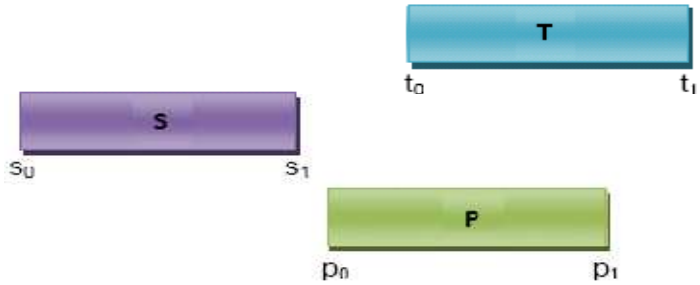


Figure 10: Cayuga event T not a successor of event S

In Figure 10 event T is not a successor of event S because event P appears in between. But here is also $s_1 < p_0 < p_1 < t_1$ correct.

In Cayuga the complex event is timestamped with the smallest interval containing the intervals of all events that make up the query result. For example, if we have a query made up of three events. And they happens at times 1, 4 and 7, then the result time stamp would have been [1, 7] [NEXT].

A thorough survey of temporal models in the CEP literature shows that there is no unique answer for choosing a successor to *A* in the figure above (Consecutiveness with interval timestamp) [NEXT, CWW].

The semantics of queries in TelegraphCQ is a bit differing from Cayuga. In TelegraphCQ for every instant in time, a set of tuples over which the query is to be executed is defined by a window on a stream. The output of a query is presented to the end-user as a sequence of sets since each execution of the query produces a set. And each set is being associated with an instant in time.

Time in TelegraphCQ is treated as a partial order in order to accommodate insecurely synchronized distributed data sources. Multiple simultaneous notions of time, such as logical sequence numbers or physical time is also allowed. An algebra with expanded relational operators exist to operate on streams and to allow a stream defined using one notion of time to be transformed into a stream using another [TCDP].

This is done by using a for-loop construct to state the sequence of windows over which the user requests the answers to the query. A variable t moves over the timeline as the for-loop iterates, and the left and right ends of each window in the sequence, and the stopping condition for the query can be defined with consideration to this variable t [NEXT].

For each stream in the query the for-loop contains the statement 'WindowIs', an input without this statement is assumed to be a static table by default. And for every group of streams that express the same window transition behavior there is one for-loop. Figure below shows the syntax of the for-loop [NEXT].

```
for(t=initial_value; continue_condition(t); change(t)){
      WindowIs(Stream A, left_end(t), right_end(t));
      WindowIs(Stream B, left_end(t), right_end(t));
      ...
}
```

Example 9: TelegraphCQ The syntax of the for-loop [NEXT]

By appropriately setting the increment statement for t in the for-loop windows can also be defined to move on-demand, or in the reverse-timestamp direction.

How both of the systems manages to capture consecutive events is determined by how they use timestamp. Where Cayuga use interval timestamp TelegraphCQ go for partial order.

3.4 Concurrency

Concurrency is about the systems capability to execute several queries at the same time, (and the possibility to interact with each other). It is also about joining the information of results from several data sources.

Cayuga's operators NEXT and FOLD reflect the concept of concurrency in the case of joining data streams. Each of these create an output stream from two input streams. By using these operators we can correlate events over two data streams thus enabling us to concurrently interact with two streams. As an example, consider two streams; S_1 and S_2 , and apply the NEXT operator to these like in 'S_1 NEXT {θ} S_2', we combine each event from S_1 with the next event in S_2 depending on if the condition {θ} is satisfied[CHP]. An example of a query with NEXT where two streams are joined with given condition:

```
SELECT *
FROM S1 NEXT {$1.T0 = T0 AND $1.T1 = T1} S2
PUBLISH NewStream
```

Example 10: Concurrency NEXT operator

'$.' indicate attribute from S_1 and '$1.' indicate attribute from S_2 . In this example we are joining the streams where the condition with timestamp is depending.

If we are interested in looking for patterns comprising two or more events we can use the FOLD operator. This operator is actually an iterated form of NEXT. The FOLD construct has the form FOLD{predExpr$_1$, predExpr$_2$, aggExpr}. These parameters consist of two conditions and in the end one aggregate computation. The first parameter describes which input events to choose in the next iteration, the second parameter implies the stopping condition for the iteration and the last parameter performs aggregate computation between iteration steps [CHP]. An example of a query with FOLD:

```
SELECT *
FROM S1 FOLD {, $1.T0 = T0 AND $1.T1 = T1 AND DUR = 1 , } S2
PUBLISH NewStream
```

Example 11: Concurrency FOLD operator

Through these operators Cayuga makes it possible to perform querying and joining of information from several data sources.

Cayuga have also another concept in concurrency, Cayuga also makes it possible to execute several queries at the same time on several streams if wanted. It is done through the execution process.

TelegraphCQ also allow performing querying and joining information from several data sources. With the possibility to execute several queries at the same

time on several streams is possible through the WITH clause in the query. To make it possible we need to limit the data sources with help of the window semantic. As mentioned earlier windowing uses a interval to bound a portion of the event stream. With the WITH clause you create new streams using the CREATE STREAM statement. Example on this:

```
drop stream streams.S1;

create stream streams.S1 (
      ID text,
      value int,
      tcqtime timestamp TIMESTAMPCOLUMN
) type unarchived;

alter stream streams.S1 add wrapper csvwrapper;

WITH
      streams.S1
      AS
      (SELECT ID, value, wtime(*)
      FROM streams.cep [RANGE BY '2 seconds' SLIDE BY '1 seconds']
      WHERE ID = 'A'
      GROUP BY ID)
```

Example 12: TelegraphCQ WITH clause

In this example the attributes is defined for the stream, further with the WITH clause it is described what the stream should contain. This stream is created of an existing stream [TWW].

3.5 Optimization

Regarding this topic we look at optimization in the contents as the process for improving a system to make some aspect of it work more efficiently or use fewer resources.

Cayuga present a feature called Multi-Query Optimization (MQO) [CWW], but any description or further information is not much to find. We assume this is a part which is still under development and therefore not focus more on this topic considering Cayuga.

TelegraphCQ have a continuously adaptive query processing mechanism eddy. Eddy is a routing operator that contains a number of modules that perform work on behalf of queries, and a number of sources that provide input data. Eddy can intercept tuples that flow into and out of these modules, observing the module behavior and choosing the order that tuples take through the modules.

The eddy obtains data from sources, determines which modules a particular tuple must visit before all processing for the tuple is complete. After a tuple has visited all required modules, it is output to all relevant result queues by the eddy. [EDY]

3.6 Cayuga vs. TelegraphCQ

Cayuga and TelegraphCQ are both systems which support on-line analysis of data streams. There are some differences in the design and implementation of these systems. We focus on topics like 1) Tuple/event definition, 2) Aggregation, 3) Consecutiveness, 4) Concurrency and 5) Optimization.

Tuple/event definitions in the systems are very similar. Both consist of id, value and timestamp(s). The difference lies in the timestamp information. Cayuga provides a start timestamp (T0) and an end timestamp (T1). In contrast TelegraphCQ only provides of a creation timestamp which is assumed to be the same as T0. With this difference Cayuga has the ability to calculate the duration of an event while TelegraphCQ does not allow events to have duration.

Events in streams occur differently in the systems. The reason is their unlike definition of timestamp(s). The end timestamp determines the order of events in Cayuga, where events with the same detection time are considered to happen simultaneously. Tuples are time-stamped by the system and are in monotonically increasing order.

As opposed to Cayuga tuples can either be externally timestamped, or timestamped by the system. Time in TelegraphCQ is considered to be partial order, since events can occur in synchronized distributed data sources. Multiple simultaneous notions of time are also allowed.

Both of the systems compare their query language with SQL which support the five standard types of aggregates; AVG, SUM, MIN, MAX and COUNT. Cayuga conveys the impression that their query language is very similar to SQL , but the aggregation is missing in CEL. In contrast TelegraphCQ truly has built in functionality like SQL for aggregations, but window semantics is required because we have endless streams of data.

Information available about Cayuga reveals that the system has no restrictions for choosing a successor of an event. Their timestamp enables freely selection of a successor among events from the stream. We assert this as an advantage since the user can define the successor based on the situation and subscriptions which mostly is varying.

To determine the successor in TelegraphCQ we are dependent on the use of push or pull applications, external or internal timestamps. But the main deciding factor is the use of the window semantics. This window semantics has a function ‘wtime(*)’ which calculates the timestamp for the last tuple. Since the tuple consists of one timestamp we know that the successor occurs after this

timestamp. We assume that in TelegraphCQ the successor can be determined by the situation and subscription.

When it comes to concurrency, Cayuga allows to execute several queries at the same time and the possibility to interact. This is done through the execution process. Cayuga also allows us to concurrently interact with two streams by using the NEXT and FOLD operators. Through these operators we can correlate events over two data streams.

TelegraphCQ also makes it possible to perform querying and joining information from several data sources. This is done through the WITH clause where you create new streams using the CREATE STREAM statement in a complex query. Even here is window semantic is needed.

Through research and study we conclude that the available information about Cayuga is sparse. Detail information and evidence is missing about some of the topics which we are interested in. For instance the Cayuga papers to some extent mention the subject optimization but no evidence or more information is to be found on their proposition.

Cayuga gives the impression of being able to solve different subscriptions through their query language. But the solutions is generally discussed in the algebra and not in the query language. This in fact reflects that Cayuga might not be on the development stage they claim to be on. We assume they have only developed and tested it on the algebra level.

TelegraphCQ manages to give evidence and information on a more detailed level in contrast to Cayuga. Example on different topics are given and development is shown through discussion. Window semantics with its different options is an example on this. Window semantic is a crucial function for TelegraphCQ and claims to make it able to solve most of the topics we are interested in.

We find the ability to define duration very useful in Cayuga, but the equivalent to it in TelegraphCQ is not clear.

On a more superior level based on research and information our opinion about DSMS and CEP is differing. We mean that it is not big differences between DSMS and CEP and will judge CEP as a part of DSMS. Description given about them both are very similar.

4

Using Cayuga and TelegraphCQ in the Home Care Scenario

This part of the paper covers the practical part. Based on the theory from previous chapters we will go through some tasks which will illustrate the topics we have discussed such as tuple definition, aggregation, consecutiveness, concurrency and optimization. We will go through these tasks and analyze and evaluate how well the systems in practice support the findings of Chapter 3 with the focus on home care scenario.

In this home care example we have a home with a few rooms that are equipped with total 5 different sensors; A, B, C, N and M. A and B read the temperature and the remaining sensors C, N and M read the motion in the home.

The goal is see how the two systems solve the same type of queries on the same event stream.

4.1 Tasks

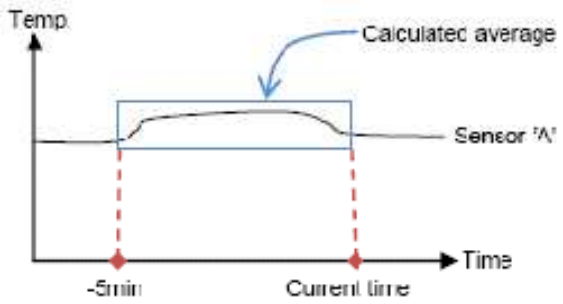
We test the two systems against a total of five tasks. But first we explain which topics each of the tasks are associated with through the table below. 'Time semantics' reflect the topic tuple definition. In Chapter 3 we discussed the difference regarding timestamp , we want to test these differences in the tasks and see if the differences matters.

Task\Topic	Time semantics	Aggregation	Consecutiveness	Concurrency	Optimization
Task 1	X	X	-	X	N/A
Task 2	X	-	X	X	N/A
Task 3	X	-	X	-	N/A
Task 4	X	-	-	X	N/A
Task 5	X	-	-	X	N/A

First we give a general description of each task and how we have interpreted it.

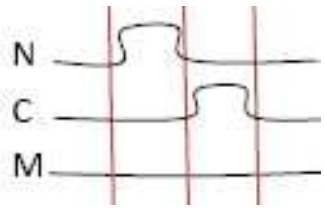
> ***Task 1:*** Tell me when the average temperature of Sensor A for the last five minutes increases more than the average temperature of Sensor B for the same time period. Assume that sensor readings from A and B are supplied by one and the same data stream".

Through this task we test the two systems ability to solve aggregation. We pick a time period of five minutes. This period consists of a time interval beginning five minutes before the current time. The Figure below shows how the time interval is defined with an example for Sensor A:



In this period the average temperature for sensor A and B has to be calculated. Further we determine if the average temperature of A is greater then the average temperature for B. If this incident occurs the system should give an output to the user.

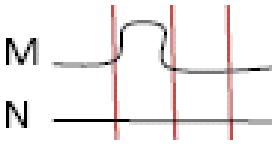
> ***Task 2:*** Tell me when Sensor N sends a reading, followed by Sensor C, while not receiving any readings from Sensor M meanwhile.



This task reflects the topic about consecutiveness and concurrency. We will through this task discover how the systems actually select one event's successor.
We divide this task into two parts for better understanding; 1) we have taken into consideration that in the first occurrence Sensor N reads value '1' and Sensor M reads value '0', while the value of Sensor C is insignificant. 2) In the next occurrence, Sensor C reads value '1' and Sensor M still reads value '0', while the value of Sensor N now is insignificant. The figure to the left shows an example of this. The vertical lines show the time interval. In this case Sensor M determines the total time this query occurs over, which starts with Sensor N and ends with Sensor C. How the events take place in relation to each other can of course be different;, it depends on the systems which we discuss later in this chapter.

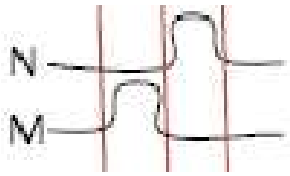
> ***Task 3:*** Tell me when Sensor N does not report anything within a minute after Sensor M has reported movement.

This task is very similar to the task above. This as well covers the topic consecutiveness. Here we look for an incident based on a preceding incident; in our case we are looking for value '0' from Sensor N based on value '1' from Sensor M. With the figure below it is a complete understanding of the task.



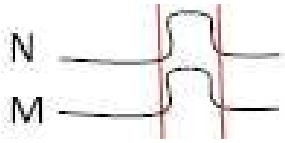
We first look for the value ’1’ from Sensor M, value from Sensor N is insignificant in that incident. When the first condition is satisfied, we now look for value ‘0’ from Sensor N. This value of ‘0’ from Senor N must occur within a minute after the first detected occurrence. The value from Sensor M is irrelevant in the second condition.

Task 4: Tell me when Sensor N and Sensor M report that one of them has registered movement.



In this task the aim is to find incident where Sensor M and Sensor N report different values at the same time. If Sensor M reports value ‘0’ then the value of Sensor N must report value ‘1’ or vice versa. Through this task we examine how concurrency is used in each of the systems.

Task 5: Tell me when both Sensor N and Sensor M read movement at the same time.



This task also examines how Cayuga and TelegraphCQ resolve concurrency. Here we are interested in incident where the both Sensor M and N report value ‘1’ at the same time.

4.2 Schema

The schema describes the format of the data occurring in the stream. The systems use the schema as input for reading the stream accordingly. The data occurring in the stream is described with attributes and the appropriate type; this can be integer, float, etc. The attributes we are interested in in our home care scenario is the id describing the sensors, a value for displaying temperature or motion and the timestamp.

4.3 Stream

We create the event stream according to the schema. Since this is only a test case we use a test file with assumed readings from the sensors. This event stream is treated by the systems as an event stream in real time.

One of the goals regarding the home care scenario and the tasks is to use the same test stream file for both. But as mentioned in the previous chapters Cayuga and TelegraphCQ read streams differently. We have still tried to focus on one stream file, where we have done adjustments as they were needed for each

system. The stream files are still very alike with regards to the data values and the timestamps.

Each event tuple in the stream reports which sensor it belongs to, the value which it has detected and timestamp. The timestamp is an example of the adjustment which has been done; the small changes are described further.

Values in the stream have been selected with the tasks in mind and in consideration to different situations which can occur. The values have been selected to test the queries utmost. The temperature sensors report temperature in integer form from value '1' up to around '40', while motion sensors report only '1' if it is any movement otherwise '0'.

Our stream consists of two kinds of timestamps, the reason is the two different systems which we want to test. In the first part of the stream you can find timestamps which indicate that the event only lasts for 1 minute. While in the last part of the stream you can find events with duration over more time. These changes have been made so we could test both system properly.

4.4 Cayuga and Home Care Application Domain

Our test case in Cayuga consists of a schema file, stream file, config files, query files and witness file.

4.4.1 Design

Each event stream has a fixed relational schema in Cayuga. The schema for our test stream is 'Sensors<ID, value, T0, T1>', below is the schema file in xml.

```
<?xml version="1.0" encoding="utf-8"?>
<StreamType xmlns="http://tempuri.org/SensorsSchema.xsd" Name="Sensors">
    <AttrNameType Name="ID" Type="string"/>
    <AttrNameType Name="Value" Type="int"/>
    <AttrNameType Name="T0" Type="int"/>
    <AttrNameType Name="T1" Type="int"/>
</StreamType>
```

Figure 11: SensorSchema.xml

The attribute ID specifies which sensor it is. The attribute value reflects the value produced by the specific sensor, T0 tells us the start time for the event and T1 the end time. The name of the stream is also included in this schema as we can see in the second line.

As previously pointed out, the stream file in Cayuga differs to some extent from the one used in TelegraphCQ. You can see in the figure below that the tuple go with the schema above, but in addition it contains the name of the stream, which

in this case is ‘Sensors’, the two last attributes represent the timestamp given internally by the system.

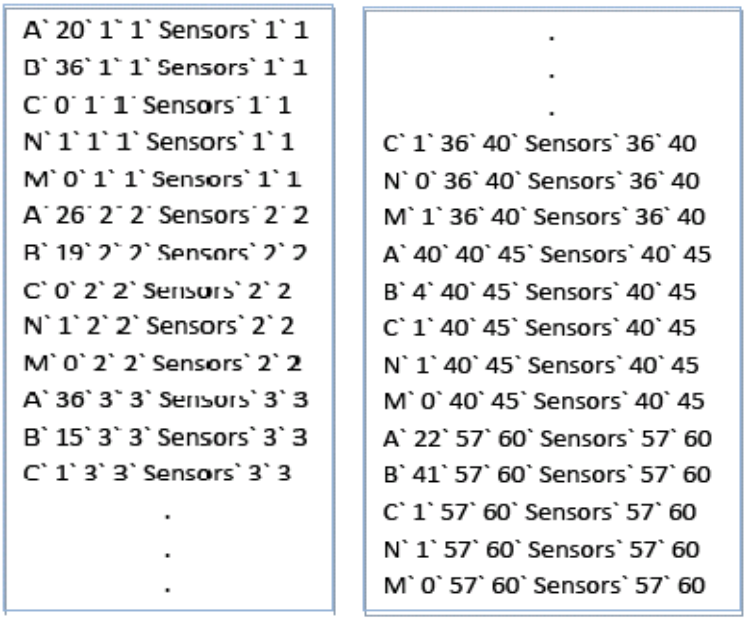


Figure 12: Cayuga SensorStream.txt

4.4.2 Test Setup

After the queries have been defined in their SQL files we need a config xml file to execute these queries. Figure below shows an example of such a config file. The important statements that we take use of are ‘QueryInputName’, ‘QueryNumber’, ‘StreamSchema’, ‘DocInputName’ and ‘DocInputStream’. Cayuga allows us to execute several queries at the same time. This can be done through using the ‘QueryInputName’ identifier. According to this we must specify the number of queries we like to execute, this can be done with ‘QueryNumber’. We must specify which schema we want to use, this must be stated with ‘StreamSchema’. In this config file we must also define which stream we are using and the name of the stream. This is defined in ‘DocInputName’ and ‘DocInputStream’.

```
<?xml version="1.0" encoding="utf-8"?>
<Config xmlns="http://tempuri.org/ConfigSchema.xsd">
 <Option Name="QueryInputMode" Value="FILE"/>
 <Option Name="AirQuery" Value="false"/>
 <Option Name="QueryInputName"
Value="req3/queries/cayuga1_1.txt;req3/queries/cayuga1_2.txt;req3/queries/c
ayuga1_3.txt"/>
 <Option Name="QueryNumber" Value="2"/>
 <Option Name="StreamSchema" Value="req3/schemas/SensorsSchema.xml"/>
 <Option Name="DocInputMode" Value="FILE"/>
 <Option Name="DocInputName" Value="req3/streams/SensorStream.txt"/>
 <Option Name="DocInputStream" Value="Sensors"/>
 <Option Name="DocNumber" Value="1"/>
 <Option Name="Verbose" Value="true"/>
 <Option Name="RecordTrace" Value="true"/>
 <Option Name="Measure" Value="true"/>
 <Option Name="CheckPointFrequency" Value="1"/>
 <Option Name="CheckPointAndTraceDir" Value="log/"/>
 <Option Name="AttrDelimiter" Value="`"/>
</Config>
```

Figure 13: Config file

When the queries have been executed the result is found in the 'witnesses.txt' file. Whenever we execute queries with the help of the config file, the result in 'witnesses.txt' will be replaced with the new result. The result in 'witnesses.txt' contains result from queries that uses the 'PUBLISH' statement.

4.4.3 Queries

Each task in this part is reflecting a complex query since we need several queries to execute one task.

Task 1:

From studies done in one of the previous chapters we discovered that Cayuga's query language does not support the five standard types of aggregates in SQL. Instead Cayuga take help of 'and', 'as', =, <=, <, > and >= to write a query from the base by defining every operator. The question is how much we have to write from the base to solve this query and if it is possible at all.

Cayuga has as also mentioned in one the of previous chapters an example of a solution for average. The example is shown in the figure below.

```
"Notify me when the difference between the current price of a stock and its 10
day movi ng average is greater than some threshold value."

SELECT Name, sum / count AS Price
FROM FILTER{DUR = 10day} (
        SELECT *, Price as sum, 1 as count FROM S
        FOLD {Name=$1.Name, , $.count + 1 as count, $.sum + Price as sum} S)
PUBLISH MovingAverage

SELECT *
FROM MovingAverage NEXT {Name = $1.Name, Price - $1.Price > threshold} S
```

If we take a look at this example they have a stream named S, with attributes name and price. We tested this query but it was difficult to get any result out of this since the query did not execute, it was too many error messages. But we will try to make a query for our task with some changes, for instance by dividing the query into several sub queries.

We begin with selecting events from sensors we are interested in. The first two sub quires indicate this, we select all events with value increasing 0 from Sensor A in one stream and in the other stream we do the same for Sensor B. In the third query we now create a new stream by joining these streams. Further we now have to calculate the values in this new stream. Since we now are interested in looking for patterns comprising two or more events we are going to use the FOLD operator. This operator is as mentioned earlier an iterated form of NEXT. We are going to use FOLD on not two different streams but on one and same stream, because we have already all the values we are interested in 'SensorsAB'. We begin with the timestamp statement, through this we are limiting the time to accrue up to five minutes. The second and third line in the FOLD statement are adding the values while the two next statements are counting the total. In the end we are calculating the average for each of the sensors.

cayuga1_1.txt:

```
SELECT *, Value AS Asum, 1 AS Acount, Value AS Aavg
FROM FILTER{ID = 'A' AND Value > 0 } Sensors
PUBLISH SensorsA
```

cayuga1_2.txt:

```
SELECT *, Value AS Bsum, 1 AS Bcount, Value AS Bavg
FROM FILTER{ID = 'B' and Value > 0} Sensors
PUBLISH SensorsB
```

cayuga1_3.txt:

```
SELECT *, Value
FROM SensorsA NEXT{ } SensorsB
PUBLISH SensorsAB
```

cayuga1_4.txt:

```
SELECT *
FROM SensorsAB FOLD{,
                    $1.T0 < $2.T1 + 5,
                    $.Asum + $2.Value AS Asum,
                    $.Bsum + $2.Value AS Bsum,
                    $.Acount + 1 AS Acount,
                    $.Bcount + 1 AS Bcount,
                    ($.Asum + $2.Value)/($.Acount + 1) AS Aavg,
                    ($.Bsum + $2.Value)/($.Bcount + 1) AS Bavg}
SensorsAB
PUBLISH SensorsABagg
```

cayuga1_5.txt:

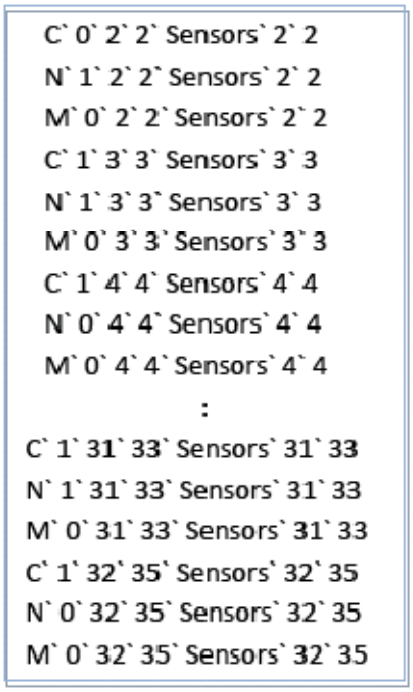
```
SELECT
FROM FILTER{DUR >= 5 AND Aavg > Bavg} SensorsABagg
PUBLISH Query1
```

We have now calculated the average for values from the two Sensors. But since we are interested in when average of the values from Sensor A increase average values from Sensor B within a time interval of five minutes we do the last query. With DUR we specify the duration which must at least be 5 minutes and average of values from Sensor A must be larger than the average of values from Sensor B within this duration.

This query did not work completely; parts of this query did work and gave results, but not the end result which we were interested in. If we only concentrate about calculating average of values from Sensor A and Sensor B each of them works, but when it comes to compare the time interval on error occurs. The reason is that we in Cayuga it is difficult or impossible to join two streams on equal

timestamps. With that problem in mind we created this query, but it still did not work properly.

Task 2:



In this task we are supposed to find out when events from Sensor C with value '1' which *follow after* events from Sensor N that also have value '1', while Sensor M should have value '0' in the mean time. So with other words we are going to discover how Cayuga actually compute consecutiveness.

As mentioned before, our stream consists of event with no duration, and some events with duration which cause overlapping. A snip of this is shown in the figure to left where we can see events with no duration in the top, and with some duration in the end.

In this stream we can see that Sensor N reports value '1' at timestamp 2 and Sensor M repots value '0'. In next timestamp we can see Sensor C report the value '1' and Sensor M still report '0'. According to this part our query is expected to among other results report timestamp [2, 3]. If we look further in this snip of the stream we can see events which are overlapping. The question now is what Cayuga will report in result for this type of incident.

We begin with creating a stream by selecting events from Sensor N with value '1'. Then we make a stream of events from C with also value '1'. Further in the third query we make a new stream, 'SensorNC', by using the FOLD operator. We choose to join the two first streams where events from Sensor C has a start timestamp later than the start timestamp of events from Sensor N, since one of our conditions is that events from Sensor C should follow events from Sensor N. We already here in our query actually decide what the successor should be. If we instead of '$1.T0 < $2.T0' write '$1.T1 < $2.T0' we exclude the last events shown in the stream figure above.

In the third query we are also defining duration on 1 minute between the events. We have to discover how this affects the last events in the stream. We had also to take events from Sensor M in consideration, so we make another stream with events from Sensor M with value '0'. We have to join this stream with the stream 'SensorNC'. Here we use the NEXT operator and join these two streams where start timestamp and end timestamp is the same.
(The Figure below shows the queries.)

```
cayuga2_1.txt:
SELECT ID, T0, T1, Value
FROM FILTER {ID = 'N' AND Value = 1} Sensors
PUBLISH SensorN

cayuga2_2.txt:
SELECT ID, T0, T1, Value
FROM FILTER {ID = 'C' AND Value = 1 } Sensors
PUBLISH SensorC

cayuga2_3.txt:
SELECT T0, T1, Value
FROM SensorN FOLD {, ( $1.T0 < $2.T0 ) as T0 AND ( $1.T1 < $2.T1 ) as T1
AND DUR = 1 ,} SensorC
PUBLISH SensorNC

cayuga2_4.txt:
SELECT ID, T0, T1, Value
FROM FILTER {ID = 'M' AND Value = 0 } Sensors
PUBLISH SensorM

cayuga2_5.txt:
SELECT T0, T1
FROM SensorNC NEXT {$1.T0 = T0 AND $1.T1 = T1} SensorM
PUBLISH Query2
```

Figure below is a snip of the result file after executing the queries above. The result file shows that the first two queries matched the right events and so did query which was interested in events with value ‘0’ from Sensor M. Further the lines in italic font shows the result from the join of events from Sensor C and Sensor N. Also this is correct according to the stream file regarding the first condition where events from Sensor C were supposed to have start timestamp later start timestamp of events from Sensor N. And if we look at the second condition which limit the duration we can see that the last events from the stream have not been matched. But if we change the duration time to for instance 3 then this query also match the last events. If we choose not to define any duration we end with many matched events which would have caused overlapping. Example on such matched event is for instance events addition to [2`3] also be [2`4] and continue from timestamp 2 till the end for instance [2`40] [2`60] and then to the same for next timestamp; [3`4], [3`5] and so on.

The lines in bold font show the end results we are interested in. The quires together have according to the stream and conditions matched on the right events. Again if we had used different time for duration it would have affected this result as well.

But if we take a closer look at the timestamp we can see that instead of for instance [2`3] it is [2`4]. Reason for this is as mentioned in task 1 is the time-lag which occur when we want to use timestamp from a stream which has been created through the FOLD operator.

```
12`WITNESS`ID`N`T0`2`T1`2`Value`1`SensorN`2`2`
12`WITNESS`ID`M`T0`2`T1`2`Value`0`SensorM`2`2`
12`WITNESS`ID`C`T0`3`T1`3`Value`1`SensorC`3`3`
12`WITNESS`ID`N`T0`3`T1`3`Value`1`SensorN`3`3`
12`WITNESS`ID`M`T0`3`T1`3`Value`0`SensorM`3`3`
10`WITNESS`T0`3`T1`3`Value`1`SensorNC`2`3`
12`WITNESS`ID`C`T0`4`T1`4`Value`1`SensorC`4`4`
12`WITNESS`ID`M`T0`4`T1`4`Value`0`SensorM`4`4`
10`WITNESS`T0`4`T1`4`Value`1`SensorNC`3`4`
8`WITNESS`T0`4`T1`4`Query2`2`4`
12`WITNESS`ID`C`T0`5`T1`5`Value`1`SensorC`5`5`
12`WITNESS`ID`N`T0`5`T1`5`Value`1`SensorN`5`5`
12`WITNESS`ID`M`T0`5`T1`5`Value`0`SensorM`5`5`
8`WITNESS`T0`5`T1`5`Query2`3`5`
12`WITNESS`ID`C`T0`7`T1`7`Value`1`SensorC`7`7`
12`WITNESS`ID`C`T0`8`T1`8`Value`1`SensorC`8`8`
12`WITNESS`ID`N`T0`8`T1`8`Value`1`SensorN`8`8`
12`WITNESS`ID`M`T0`8`T1`8`Value`0`SensorM`8`8`
12`WITNESS`ID`C`T0`9`T1`9`Value`1`SensorC`9`9`
10`WITNESS`T0`9`T1`9`Value`1`SensorNC`8`9`
12`WITNESS`ID`M`T0`10`T1`10`Value`0`SensorM`10`10`
8`WITNESS`T0`10`T1`10`Query2`8`10`
                    :
```

Task 3:

```
N` 0` 6` 6` Sensors` 6` 6
M` 1` 6` 6` Sensors` 6` 6
N` 0` 7` 7` Sensors` 7` 7
M` 1` 7` 7` Sensors` 7` 7
N` 0` 9` 9` Sensors` 9` 9
M` 1` 9` 9` Sensors` 9` 9
N` 0` 11` 11` Sensors` 11` 11
M` 1` 11` 11` Sensors` 11` 11
N` 0` 13` 13` Sensors` 13` 13
M` 1` 13` 13` Sensors` 13` 13
N` 0` 14` 14` Sensors` 14` 14
M` 1` 14` 14` Sensors` 14` 14
N` 0` 36` 40` Sensors` 36` 40
M` 1` 36` 40` Sensors` 36` 40
```

This task can resemble Task 2. In this task we only care about values from Sensor M and N and drop the part with Sensor C from the task above. We want the events from Sensor N where the value is ‘0’ and from Sensor M we want the events which report value ’1’ in the same time interval.

The figure at the left is a snip of the stream file where we have picked out events from Sensor N and M where they matches to the tasks requirement. As we can see the last event has duration over 1minute.

We start with creating a stream with events from Sensor N which have the value ‘1’ and

another stream with events from Sensor M which have the value ‘0’. We call these streams ‘SensorN’ and ‘SensorM’.
Further we are now joining these streams with the conditions which fulfill the requirement of this task. As pointed out we want only the events where they report the interested values at the same time interval. The queries are shown in the figure below.

```
cayuga3_1.txt:
SELECT *
FROM FILTER {ID = 'M' AND Value = 1} Sensors
PUBLISH SensorM

cayuga3_2.txt:
SELECT *
FROM FILTER {ID = 'N' AND Value = 0 } Sensors
PUBLISH SensorN

cayuga3_3.txt:
SELECT *
FROM SensorM FOLD {, $1.T0 = $.T0 AND $1.T1 = $.T1 AND DUR <= 4 ,}
SensorN
PUBLISH Query3
```

Because of the latter requirement which has been mentioned in our third query we are defining both timestamps to be equal. In the end we are also defining the duration which is limited up to 4 minutes. The reason for this limitation is to avoid all the overlapping events as also explained in the previous tasks. The result of this query is shown in the figure below.

```
12`WITNESS`ID`N`Value`0`T0`6`T1`6`SensorN`6`6`
12`WITNESS`ID`M`Value`1`T0`6`T1`6`SensorM`6`6`
12`WITNESS`ID`N`Value`0`T0`7`T1`7`SensorN`7`7`
12`WITNESS`ID`M`Value`1`T0`7`T1`7`SensorM`7`7`
20`WITNESS`ID_1`M`Value_1`1`T0_1`6`T1_1`6`ID`N`Value`0`T0`7`T1`7`Query3`6`7`
12`WITNESS`ID`N`Value`0`T0`9`T1`9`SensorN`9`9`
12`WITNESS`ID`M`Value`1`T0`9`T1`9`SensorM`9`9`
20`WITNESS`ID_1`M`Value_1`1`T0_1`7`T1_1`7`ID`N`Value`0`T0`9`T1`9`Query3`7`9`
12`WITNESS`ID`N`Value`0`T0`10`T1`10`SensorN`10`10`
20`WITNESS`ID_1`M`Value_1`1`T0_1`9`T1_1`9`ID`N`Value`0`T0`10`T1`10`Query3`9`10`
12`WITNESS`ID`N`Value`0`T0`11`T1`11`SensorN`11`11`
12`WITNESS`ID`M`Value`1`T0`11`T1`11`SensorM`11`11`
12`WITNESS`ID`N`Value`0`T0`12`T1`12`SensorN`12`12`
20`WITNESS`ID_1`M`Value_1`1`T0_1`11`T1_1`11`ID`N`Value`0`T0`12`T1`12`Query3`11`12`
12`WITNESS`ID`N`Value`0`T0`13`T1`13`SensorN`13`13`
12`WITNESS`ID`M`Value`1`T0`13`T1`13`SensorM`13`13`
12`WITNESS`ID`N`Value`0`T0`14`T1`14`SensorN`14`14`
12`WITNESS`ID`M`Value`1`T0`14`T1`14`SensorM`14`14`
20`WITNESS`ID_1`M`Value_1`1`T0_1`13`T1_1`13`ID`N`Value`0`T0`14`T1`14`Query3`13`14`
12`WITNESS`ID`N`Value`0`T0`28`T1`29`SensorN`28`29`
12`WITNESS`ID`N`Value`0`T0`32`T1`35`SensorN`32`35`
12`WITNESS`ID`N`Value`0`T0`36`T1`40`SensorN`36`40`
12`WITNESS`ID`M`Value`1`T0`36`T1`40`SensorM`36`40`
```

We expected answers like [6`7], [7`9], [11`11], [13`14] and [36`40] based on the stream file shown in the beginning of this task. These answers compared to the result file shown above only events in timestamp [6`7], [7`9] and [13`14] are similar. Timestamps which do not properly agree with the conditions given in the third query are [9`10], [11`12] and the missing event in time interval [36`40].

Assumptions for why our query reported [9`10], [11`12] instead of the time interval [11`11] could be the time-lag which occur when joining two streams. Ergo for the same reason as in the previous tasks.

Through modifying the query by changing the timestamp definition in the third query we have discovered that the reason can also be how Cayuga read the events. For instance if we look at the line in bold font where ‘query3’ has reported the interval [7`9] Cayuga read event from ‘SensorN’ with value ‘0’. Therefore ‘query3’ reports [9`10] in the next line, since the timestamp condition is fulfill because of the time-lag.

Cayuga does not report an event in the time interval [36`40]. We try to modify the duration interval from 4 to for instance 5 but it still do not report this time interval. We modified the stream file to test out possible reason for why Cayuga is not reporting this event. After the time interval [36, 40] the next event comes in the interval [40`45]. In this next time interval we changed the event where M was reporting value ‘1’ and Sensor N was reporting value ‘0’. The reason for this was

because of the time-lag which happens. We expected Cayuga to report [36, 45] or [40`45]. But these changes did not affect the result, which still reported the same as the result file shown in this task.

Task 4:

```
N`1`1`1`Sensors`1`1
M`0`1`1`Sensors`1`1
N`1`2`2`Sensors`2`2
M`0`2`2`Sensors`2`2
N`1`3`3`Sensors`3`3
M`0`3`3`Sensors`3`3
N`0`4`4`Sensors`4`4
M`0`4`4`Sensors`4`4
N`1`5`5`Sensors`5`5
M`0`5`5`Sensors`5`5
```

The goal of this task is to make a query which will tell us when only one of Sensor N *or* Sensor M reports movement, i.e. value '1'. The Figure to left shows a small part of the stream where we have picked out only events from Sensor N and M. In this figure we can see that from timestamp 1 including timestamp 3 only one of the sensors report value '1' until timestamp 4, then both of the sensors report value '0'.

The simplest thing we could do is to make a new stream which consists of every event Sensor N has reported in the stream, we do not bother about the value now. We do the same with events from Sensor M.

Further we might have made a query where we could have joined these streams with the condition that the value was not similar. But Cayuga does not have the operator which indicates 'not equal'.

Instead we try to solve this query by stating streams like the following. We first create streams 'SensorN' and 'SensorM'. These streams consist every event which has been reported from the belonging Sensor. Further we in the third query join these streams with the FOLD operator. Figure below shows the third query, on which conditions we have tried to join the two streams.

cayuga4_1.txt:

```
SELECT ID, Value, T0, T1
FROM FILTER {ID = 'M' AND (Value < 2) AS Value} Sensors
PUBLISH SensorM
```

cayuga4_2.txt:

```
SELECT ID, Value, T0, T1
FROM FILTER {ID = 'N' AND (Value < 2) AS Value } Sensors
PUBLISH SensorN
```

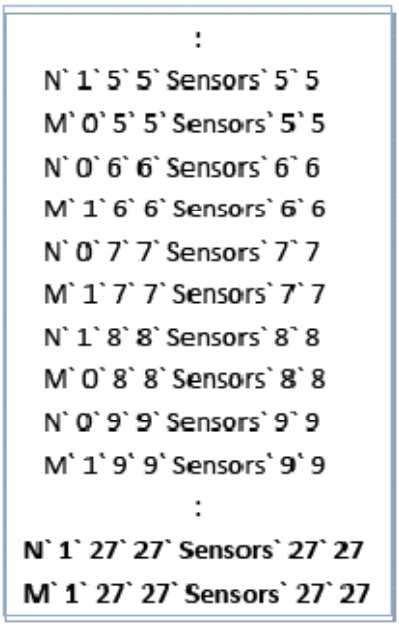
cayuga4_3.txt:

```
SELECT *
FROM SensorN FOLD { , $1.T0 = $2.T0 AND $1.T1 = $2.T1  AND
$1.Value + $2.Value = 1 , } SensorM
PUBLISH Query4
```

In the last query we tried to add the values from stream ‘SensorN’ with the values from stream ‘SensorM’ where the result should be 1. The thought behind this was that since we are only interested in events where one of them had the value’1’, than the other naturally had to report value ‘0’. Adding the values together would have been 1.

But this query did not report any value at all. It has been modified many times, for instance using NEXT instead of FOLD. The reason for this is the operators cause to time-lag will this always be evaluated to FALSE and no result will come.

Task 5:



With this last task we want to know when both sensor N and M report movement at the same time. The lines in bold font in the figure to the left show where in the stream this happens.

To match this event with our query we begin with creating two streams. One for events from Sensor M and one for events from Sensor N. Both of the streams contain events which reported value ‘1’.

Further we now join these two streams on the condition that their timestamps are equal. Because of previous tasks we are now familiar that if we do not limit this with duration we will end up with many event combinations. Therefore we limit our query with duration on 1 minute. The Figure below shows how our queries look like.

cayuga5_1.txt:

```
SELECT ID, Value, T0, T1
FROM FILTER {ID = 'N' AND Value = 1} Sensors
PUBLISH SensorN
```

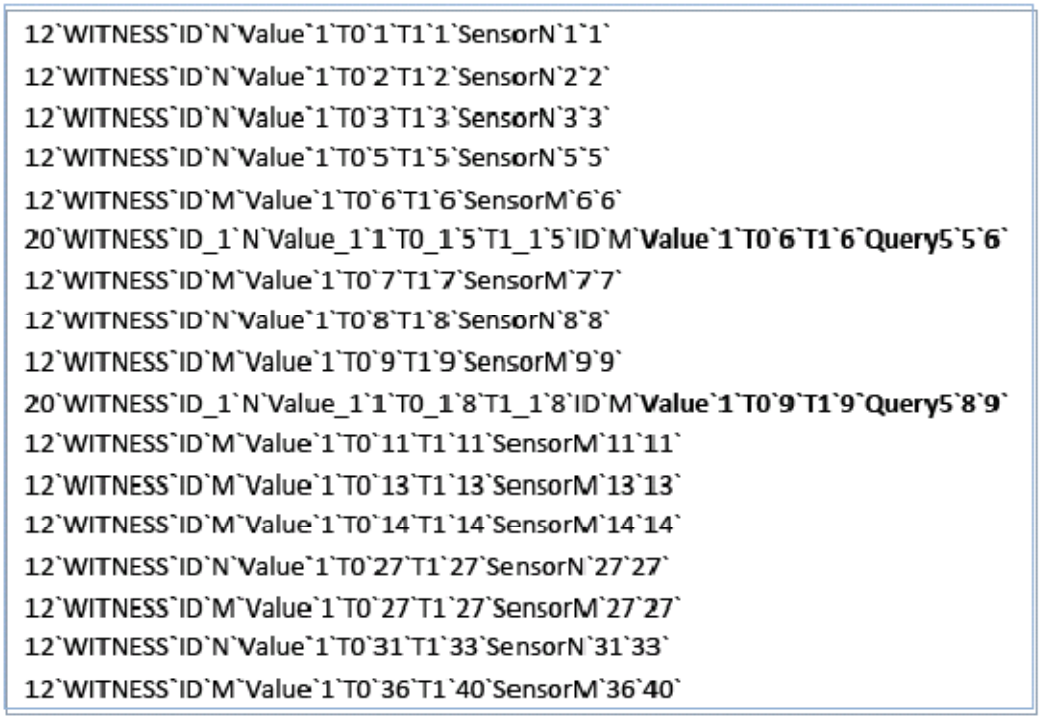
cayuga5_2.txt:

```
SELECT ID, Value, T0, T1
FROM FILTER {ID = 'M' AND Value = 1} Sensors
PUBLISH SensorM
```

cayuga5_3.txt:

```
SELECT *
FROM SensorN FOLD {, $1.T0 = T0 AND $1.T1 = T1 AND DUR = 1 , } SensorM
PUBLISH Query5
```

Through the previous tasks we have also noticed the time-lag and what is leaded to. Therefore even this time our result is differing from the expected result. Result is shown in the figure below.



The query matched on event in time interval [5`6] and [8`9], which is not what we expected according to our stream file. But if we look back to the snip from the stream which was shown in the starting of this task, we can see that the interval which is reported in the result file does cover the timestamps we was expecting, but has instead merged it. In the interval [5`6] and [8`9] is both of the sensors reporting value ‘1’. So with the time-lag explanation this query is actually reporting the correct time interval. But if we take a closer look to the snip of the stream file, we can see that Sensor M reports value ‘1’ with the timestamp 7. So the question is why this timestamp is not included in the result file which is expected since timestamp 8 which is the next event report value ‘1’ from Sensor N. If we had used NEXT this timestamp would also have been included, but with many combinations of time interval. The reason for why this is not included with this query might be how Cayuga read the events. Because if Cayuga have compared events from time interval 7 with the previous time interval 6, then it is correct that our query does not match.

4.5 TelegraphCQ and Home Care Application Domain

Our test case in TelegraphCQ consists of schema file, stream file, query files and output files.

4.5.1 Design

The schema in TelegraphCQ is defined in a SQL file, shown in the figure below. In this file we create a stream by giving the stream a name, in this case ‘streams.cep’, further we declare the attributes we are interested in. The attributes we use are ID, value, t0, t1 and tcqtime timestamp.

```
drop schema streams;
create schema streams;

drop stream streams.cep;

create stream streams.cep (
        ID text,
        value int,
        t0 int,
        t1 int,
        tcqtime timestamp TIMESTAMPCOLUMN
) type unarchived;
```

Figure 14: TelegraphCQ schema.sql

As we recall from previous chapters TelegraphCQ has only one timestamp, t0. But since we want to use the same stream for both of the systems we define ‘t1’ also. The statement ‘tcqtime timestamp TIMESTAMPCOLOUMN’ gives the current time, this time is set externally.

Stream in TelegraphCQ will accordingly to the schema contain <ID, value, t0, t1>. You can see in the figure below how the stream file in TelegraphCQ is.

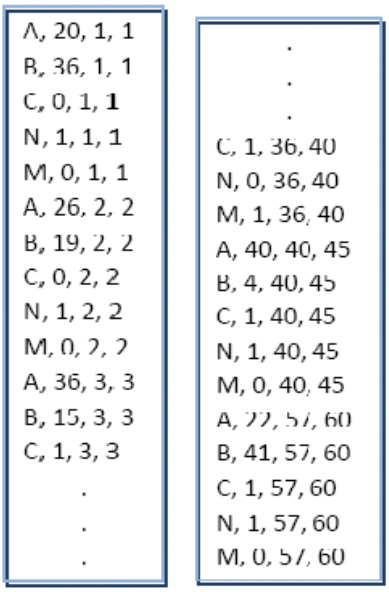


Figure 15: TelegraphCQ Stream.txt

As we can see this stream file does not contain the stream name and extra timestamp attributes as seen in the stream file for Cayuga. But otherwise we have used the same values and timestamps.

4.5.2 Test Setup

Compared to Cayuga only one query can be executed at a time. The queries are executed with the help of two main commands that are executed in each terminal. Example of the commands:

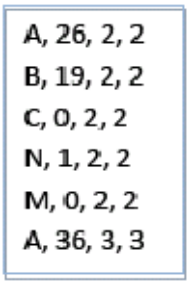
Terminal 1:PSQL.sh <name of the query file> <selected output file>
Terminal 2: cat <name of the stream file> | source.pl localhost 9556 csvwrapper,< name of the stream> <sleep time> <lines per sleep time>

(We have executed our queries with sleep time 1 and lines per sleep time 5).

4.5.3 Queries

This chapter will answer the tasks above. Queries made to solve the tasks will be shown with the result compared to the result we were expecting. In this chapter we will also explain why particular task is associated to the particular topic which was presented earlier.

Task 1:



Since this is a test we have replaced five minutes with two seconds in this task. To the left we have a snip of the stream file. Our query should give the first output from this part of the stream since in TelegraphCQ time stamping happen internal . We are now on timestamp 2 and 3. On two second the query manages to read two events from each sensor. Here in the stream is the first time average of values from Sensor A increase the average from values of Sensor B. In the pervious timestamp average of values from Sensor B was increasing.

The query solved for this in TelegraphCQ starts with creating two new streams. One for the calculated average of value from Sensor A and one for the calculated average of value from Sensor B. With the WITH clause we have defined what the stream should consist of. TelegraphCQ makes it possible to let the user use the AVG operator. But if we use this operator without the window clause, we end up with error messages. The query will not execute without the window clause. Since the task asks for the last five minutes (two seconds) we define the window clause with ‘RANGE BY’ and with the ‘SLIDE BY’ statement to make sure it will be the *last* minutes (seconds).

```
drop stream streams.avgA;

create stream streams.avgA (
      ID text,
      averageA float,
      tcqtime timestamp TIMESTAMPCOLUMN
) type unarchived;

alter stream streams.avgA add wrapper csvwrapper;

drop stream streams.avgB;

create stream streams.avgB (
      ID text,
      averageB float,
      tcqtime timestamp TIMESTAMPCOLUMN
) type unarchived;

alter stream streams.avgB add wrapper csvwrapper;

WITH
      streams.avgA
      AS
      (SELECT ID, AVG(value) as averageA, wtime(*)
      FROM streams.cep [RANGE BY '2 seconds' SLIDE BY '1 seconds']
      WHERE ID = 'A'
      GROUP BY ID)

      streams.avgB
      AS
      (SELECT ID, AVG(value) as avgerageB, wtime(*)
      FROM streams.cep [RANGE BY '2 seconds' SLIDE BY '1 seconds']
      WHERE ID = 'B'
      GROUP BY ID)

(SELECT A.ID, A.averageA, B.ID, B.averageB, A.tcqtime
FROM streams.avgA AS A,
   streams.avgB AS B
WHERE A.tcqtime = B.tcqtime AND A.averageA > B.averageB);
```

In the end we join these two streams where the time matches and the average of Sensor A exceeds the average of Sensor B.

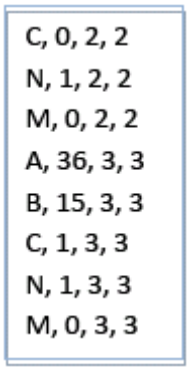
```
A,31,B,17
A,35.5,B,21
A,31.5,B,20.5
A,32.5,B,18
A,26,B,20
A,24.5,B,21
A,24,B,22
A,25.5,B,5
```

The result shown in the figure on left side is from the query. This reflects the expected result and we assume this to be the right answer. The result shows the average of Sensor A over the given time in window clause where it increases the average of Sensor B.

This query and its output show that TelegraphCQ allow us to perform aggregate operators on streams, the only thing which

is a must is the use of windowing. We also have to create new streams because this task required calculating average for two sensors. If it has been a simple task it might have not been necessary.

Task 2:



In TelegraphCQ an event contains only one timestamp, for this reason we can not know the duration of an event. We can assume it from the timestamp of the next event, but the time difference can also mean that the particular sensor has not reported anything in that time duration. For that reason as mentioned before our stream with consideration to TelegraphCQ consist of no duration time between t0 and t1. But because of this we avoid overlapping when events occur. Events with the equal timestamp occur with different sensors id so every tuple is unique this way.

The Figure to left is a snip out of the stream file. The query should first notice this part of the stream. At the first timestamp Sensor N is reading value'1', while Sensor C and Sensor M reads value '0'. In the next timestamp it is a "hit", Sensor C has reported value '1' and Sensor M is still reporting value '0'.

Solving this query we have in mind that the following event from Sensor C has a timestamp later than event from Sensor N and the condition will restrict it to be the first next event. The condition is of course the value which the sensors reports. But how to define "*meanwhile*" is the challenge since events in TelegraphCQ does not have any duration time. A attempt to solve this could be the use of T1, but it would not work either also because it is no duration time.

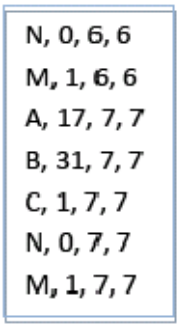
If we declare that T0 for events from Sensor M is the same as T0 for events from Sensor N, and T0 for events from Sensor C start later than T0 for events from Sensor N, we can not state that timestamp for Sensor C and M should also be he same. It can only be equal with one of them. The query was solved as following:

```
drop stream streams.readN;
create stream streams.readN (
        ID text,
        value int,
        t0 int,
        tcqtime timestamp TIMESTAMPCOLUMN
) type unarchived;
alter stream streams.readN add wrapper csvwrapper;

drop stream streams.readC;
create stream streams.readC (
        ID text,
        t0 int,
        value int,
        tcqtime timestamp TIMESTAMPCOLUMN
) type unarchived;
alter stream streams.readC add wrapper csvwrapper;

drop stream streams.readM;
create stream streams.readM (
        ID text,
        t0 int,
        value int,
        tcqtime timestamp TIMESTAMPCOLUMN
) type unarchived;
alter stream streams.readM add wrapper csvwrapper;

WITH
        streams.readN  AS
        (SELECT ID, t0, value, wtime(*)
        FROM streams.cep [RANGE BY '1 seconds' SLIDE BY '1 seconds']
        WHERE ID = 'N' AND value = '1'
        GROUP BY ID, t0, value)

        streams.readC  AS
        (SELECT ID, t0, value, wtime(*)
        FROM streams.cep [RANGE BY '1 seconds' SLIDE BY '1 seconds']
        WHERE ID = 'C' AND value = '1'
        GROUP BY ID, t0, value)

       streams.readM  AS
        (SELECT ID, t0, value, wtime(*)
        FROM streams.cep [RANGE BY '1 seconds' SLIDE BY '1 seconds']
        WHERE ID = 'M' AND value = '0'
        GROUP BY ID, t0, value)

(SELECT C.ID, C.value,C.t0, N.ID, N.value, N.t0, M.ID, M.value, M.t0
FROM  streams.readN AS N,
     streams.readC AS C,
     streams.readM AS M
WHERE N.value = C.value AND N.t0 < C.t0 AND N.t0 = M.t0 );
```

As we can see we have created three streams, first with value ‘1’ from Sensor N, second with value ‘1’ from Sensor C and the third stream with value ‘0’ from Sensor M. In the end we have to join it right.

Our query did not report correct result, when it came to events from Sensor M.

Task 3:



We can see on the left values from sensors N and M from a snip of the stream file. The first hit from the query should be from this part of the stream. The first time Sensor M reports value ‘1' is at timestamp 6, and since we are interested in value ‘0’ from Sensor N *after* the M has reported, the “hit” should be then value reported from Sensor N next at the next timestamp, in our case timestamp 7.

This task as well as the task above will test consecutiveness and see how it actually works. We start with creating two new streams, one with value ‘1’ from Sensor M and one with value ‘0’ from Sensor N. The challenge here is how to define “*within a minute after*”. It is implicitly that t0 for Sensor N has to be later than t0 for Sensor M. But we have to define a limit for timestamp y0 from Sensor M. Below is the solution we have used to hit on the right answer.

```
drop stream streams.repM;

create stream streams.repM (
      ID text,
      value int,
      t0 int,
      tcqtime timestamp TIMESTAMPCOLUMN
) type unarchived;

alter stream streams.repM add wrapper csvwrapper;

drop stream streams.repN;

create stream streams.repN (
      ID text,
      value int,
      t0 int,
      tcqtime timestamp TIMESTAMPCOLUMN
) type unarchived;

alter stream streams.repN add wrapper csvwrapper;

WITH

      streams.repM
      AS
      (SELECT ID, value, t0, wtime(*)
      FROM streams.cep [RANGE BY '1 seconds' SLIDE BY '1 seconds']
      WHERE ID = 'M' AND value = '1'
      GROUP BY ID, value, t0)

      streams.repN
      AS
      (SELECT ID, value, t0, wtime(*)
      FROM streams.cep [RANGE BY '1 seconds' SLIDE BY '1 seconds']
      WHERE ID = 'N' AND value = '0'
      GROUP BY ID, value, t0)

(SELECT M.ID, M.value, M.t0, N.ID, N.value, N.t0
FROM streams.repM AS M,
   streams.repN AS N
WHERE M.t0 + 1 = N.t0 AND N.value < M.value);
```

```
M,1,6,N,0,7
M,1,9,N,0,10
M,1,11,N,0,12
M,1,13,N,0,14
M,1,31,N,0,32
```

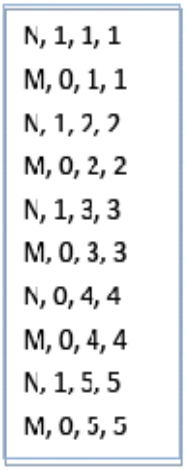
The last WHERE clause defines the limit for timestamp t0 for Sensor N. Since we are using integer format for our timestamp, this allows us to write the query like this.

The result from the query, and which reflects the expected result is shown on the left side. We can see from the value with the timestamp that the result according to the stream is correct.

If we look at the last line in the result figure, the query has also managed to match on events with duration. The timestamp for event from sensor M is (31, 33). And the timestamp for the event from Sensor N is (32, 34). From this result

we can then assume that next event in TelegraphCQ is an event which has the start timestamp after its start timestamp of the first event.

Task 4:

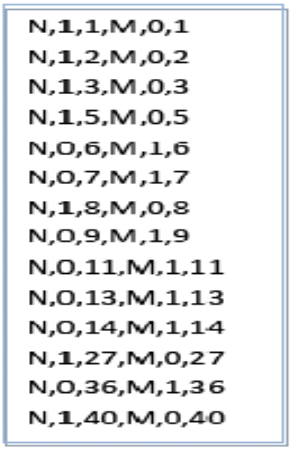


This task is meant to reflect concurrency. We will through this task see if TelegraphCQ manages to solve this kind of tasks. Let us see on the figure to left which contain some few values from Sensor N and M got from the stream file. We can see that until timestamp 4 one of the report movement by reading value '1'. So our query should not report timestamp 4 considering this part of the stream.

The query we defined begins with creating two streams, one with all the values from Sensor N and the other with all the values from Sensor M. It is in the last SELECT clause we talk about the values when joining the two streams together as seen in the figure below.

```
drop stream streams.onlyN;

create stream streams.onlyN (
	ID text,
	t0 int,
	value int,
	tcqtime timestamp TIMESTAMPCOLUMN
) type unarchived;

alter stream streams.onlyN add wrapper csvwrapper;

drop stream streams.onlyM;

create stream streams.onlyM (
	ID text,
	t0 int,
	value int,
	tcqtime timestamp TIMESTAMPCOLUMN
) type unarchived;

alter stream streams.onlyM add wrapper csvwrapper;

WITH
		streams.onlyN
		AS
		(SELECT ID, t0, value, wtime(*)
		FROM streams.cep [RANGE BY '1 seconds' SLIDE BY '1 seconds']
		WHERE ID = 'N'
		GROUP BY ID, t0, value)

		streams.onlyM
		AS
		(SELECT ID, t0, value, wtime(*)
		FROM streams.cep [RANGE BY '1 seconds' SLIDE BY '1 seconds']
		WHERE ID = 'M'
		GROUP BY ID, t0, value)

(SELECT N.ID, N.value, N.t0, M.ID, M.value, M.t0
FROM streams.onlyN AS N,
	streams.onlyM AS M
WHERE N.t0 = M.t0 AND M.value <> N.value);
```

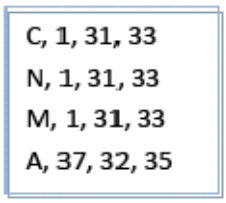
In the last WHERE clause we define that the timestamp from each stream should be the same when their values differ. With other words, when their values don't show the same values, either if it is the value '1' or value '0'.



The result reflects the correct assumed answer and is shown in the figure to the left. The result shows us the value with the timestamp. We can see here by comparing it with the stream file that they have manage to skip timestamp 4 which would have been wrong.

This task shows us that TelegraphCQ can manage to join and match on concurrency.

Task 5:



This task is very alike the task above, but now we have to search in the stream where they have the same value as well as timestamp. The Figure to the left shows us a snip of the stream file where the query should hit on. This is the only hit, since this is the only place in the stream where Sensor N and M report value '1' at the same time.

We could have changed only the last line in the query above, but chose to create new streams of Sensor N and Sensor M which only contains value '1'. In the end we joined the two streams where the timestamp was the same.

```
drop stream streams.moveN;

create stream streams.moveN (
      ID text,
      t0 int,
      value int,
      tcqtime timestamp TIMESTAMPCOLUMN
) type unarchived;

alter stream streams.moveN add wrapper csvwrapper;

drop stream streams.moveM;

create stream streams.moveM (
      ID text,
      t0 int,
      value int,
      tcqtime timestamp TIMESTAMPCOLUMN
) type unarchived;

alter stream streams.moveM add wrapper csvwrapper;

WITH
            streams.moveN
            AS
            (SELECT ID, t0, value, wtime(*)
            FROM streams.cep [RANGE BY '1 seconds' SLIDE BY '1 seconds']
            WHERE ID = 'N' AND value = '1'
            GROUP BY ID, t0, value)

            streams.moveM
            AS
            (SELECT ID, t0, value, wtime(*)
            FROM streams.cep [RANGE BY '1 seconds' SLIDE BY '1 seconds']
            WHERE ID = 'M' AND value = '1'
            GROUP BY ID, t0, value)

(SELECT N.ID,N.value, N.t0, M.ID, M.value, M.t0
FROM streams.moveN AS N,
   streams.moveM AS M
WHERE N.t0 = M.t0);
```

The result according to the stream was correct. This is evidence on how TelegraphCQ manages to join two streams which are created from one and the same stream and hits on the demanded event.

```
N,1,31,M,1,31
```

4.6 Conclusion

The tasks have given us insight on how both Cayuga and TelegraphCQ manage and resolve the different topics. The table below is a general view describing with the signs; + and – if the system have managed to solve the topics or not respectively. There are some topics which has been resolved partly this is pointed out with +/-.

Topic\System	Cayuga	TelegraphCQ
Time semantics	+	-
Aggregation	-	+
Consecutiveness	+/-	+
Concurrency	+/-	+
Optimization	-	+

Tuple definition in Cayuga consists of id; indicating the sensor it belongs to, value, start timestamp (T0) and end timestamp (T1). The last mentioned has been an advantage in Cayuga. Since Cayuga has two timestamps it is possible to calculate duration, in addition to have a built in operator like DUR to also calculate duration. To have the ability to calculate the duration became important in tasks where we had to take into consideration of terms like; same time period, meanwhile, within a minute and same time.

Tuples in TelegraphCQ compared to Cayuga originally do not have an end timestamp (T1). Truly we have used T1 in our test stream file, but an event in TelegraphCQ has no duration.

If it is important to know the duration for one particular event, Cayuga has the possibility to answer this question because of T1, compared to the originally tuple definition of TelegraphCQ.

Aggregation is attempted in task 1 where the systems are supposed to resolve average. TelegraphCQ solve this task by allowing the use of aggregation functions, the syntax is alike SQL. The only thing which is demanded is the use of window semantic which is needed because we are in general facing infinite streams in real-time.

Compared to TelegraphCQ, Cayuga does not support any of the five standard types of aggregates and does not have any other built in functions to replace them neither. Aggregates can however be constructed without the built in functions. But their query fails and neither does it work with sub queries where we

divide the whole query in several parts. An attempt to begin from the base and define the functions does not work either. One of the problems which occurs when we divide the query or start from the bottom is the time-lag which after some queries is difficult to have control over.

In task 2 which reflect both consecutiveness and concurrency, Cayuga manage to report a result, but because of the time-lag it is not the exact expected result. In Cayuga time-lag occurs when for instance creating a new stream with the help of FOLD or NEXT. In addition to time-lag Cayuga also merges for instance events from two to one. If you want events with duration 1, Cayuga do reports more than demanded. It does not have to be the totally wrong result, but more than the expected result.

This incident also applies to the next task which tests the consecutiveness topic. Cayuga makes new timestamps after joining two streams with the FOLD or NEXT operator. To use these timestamps further is almost impossible. An attempt on this has also been tested by renaming the timestamp in the SELECT statement when joining. But to use these new renamed timestamp is giving no result.

The last two tasks where we try to resolve concurrency Cayuga does not report anything in task 4. Cayuga describes its query language as SQL-like, but it not only does not support the five standard aggregates, but also for instance does not have <>. Many attempts were done, but none of the queries did report anything near the correct answer or did not report any answer at all. But in the last query Cayuga reported result which was very close to the expected result. As mentioned earlier time-lag occurs and events are been merged. In this task several events have been merged to one. This is not a problem if they do not contain timestamps where the condition which has been set is not fulfilled, but which happen. In addition to the extra timestamp which had been added in the most incident the last events which should be included in the end result is missing.

In contrast to Cayuga TelegraphCQ reports the expected results for every task except the second task. The second task reflects the topic consecutiveness and concurrency together. TelegraphCQ did resolve tasks where it was focus on only one of the topics without making very complex queries. But due to the second task we discovered that TelegraphCQ does not manage to resolve concurrency and consecutiveness together, where it has to take care of three events timestamp. As mentioned, TelegraphCQ does not consider events to have a duration, which might have solved this task. Then we could have defined duration over several time intervals when joining streams.

5

Conclusion and Future Work

In this thesis we discover the differences and similarities between Cayuga and TelegraphCQ. To underline these differences and similarities we focus on vital and relevant topics. Through comparing these two systems we also discover if they are capable of being used as a technical solution in a home care environment, where the outcome is crucial.

The comparison is done in two parts; theoretical and practical. The theoretical part focus on topics like: focus on the topics; tuple/event definition, aggregation, consecutiveness, concurrency and optimization.

The practical part is based on the theoretical part. In this part we go through five relevant tasks and analyze and evaluate how well the systems in practice can be used in a home care environment. The goal is see how the two systems solve the same type of queries on the same event stream.

Through research and study we conclude that the available information about Cayuga is sparse. Detailed information and evidence is missing about some topics which we are interested in. Cayuga gives the impression of being able to solve different subscriptions through their query language, but most of the solutions are given only in algebra. When testing these statements in the practical part it turns out that the query language is not capable of doing what it has been proposed to do in the theoretical part. This in fact reveals that Cayuga might not be on the development stage it claims to be on. For this reason none of the five tasks reported results as expected or executed completely.

In the theoretical part we also discovered that Cayuga merges time intervals when joining events with different timestamps. This we experienced in the practical part as varying. In some incidents it did apply and some not. The incidents which it applied for anyhow did not merge the right timestamps. Time-lag occurred during the joining. Also this caused to not provide the expected results.

Overall the query language of Cayuga did not work as described in the theoretical part. The expected result based on the given statements in the available information differs and reflect lack of ability to complete different tasks.

But we find the ability to define duration very useful in Cayuga. This enables us to define consecutiveness and concurrency in a query. But as mentioned the main problem is how streams are joined.

TelegraphCQ manages to give evidence and information on a more detailed level in contrast to Cayuga. Examples on different topics are given and development is shown through discussion and founding. Window semantics with its different options is an example on this. Window semantics is a crucial function for TelegraphCQ and claims to make it able to solve most of the topics we are interested in. This was proven in the practical part, where almost every task completed with the correct expected result.

TelegraphCQ completed the task with not much complexity. The query containing both consecutiveness and concurrency showed to be too much to handle for TelegraphCQ and the query failed. Tuples containing a real end timestamp , thus describing the events duration, could have been utilized to provide the expected correct result.

Some of the most important criteria for a system in a home care environment are to report the correct value and time in addition to manage large amount of data. In consideration of this we conclude that compared to Cayuga, TelegraphCQ is the most capable system for the home care domain. The reason for this is that Cayuga does not report the correct value at times and neither the time interval. And part of the query language is insufficient and too complex to perform correctly.

TelegraphCQ in contrast reports the correct value and time interval with the possibility to use built in functions from SQL, and does not demand queries with lot of complexity. If TelegraphCQ also could support the duration function as Cayuga it might have resolved the task where it failed. It is difficult to find a replacement for the duration function in TelegraphCQ. Another solution is to have the possibility to define duration through an end timestamp defined explicitly in the tuples.

TelegraphCQ compared to Cayuga with focus on the topics with in the home care domain is a more capable system to use. Implementation to execute queries with both consecutiveness and concurrency could be a plus for TelegraphCQ. It is now only possible to determine the successor with foundation on start timestamp, but we miss the ability to determine the successor based on the end timestamp.

During the discussions in this thesis we could see that there is much more to do and learn. In the beginning of this thesis the first question which arise is if it is big dissimilarity between CEP and DSMS. Our opinion is that CEP might be a DSMS but more investigation and research is needed to assume this completely.

Optimization was a topic which is not much discussed in this thesis because of the limited information. By analyzing Cayuga's algebra more in the depth we

assume many unanswered question will be answered. But it is a process in itself to translate the algebra information into the query language level.

The duration was a drawback in TelegraphCQ, to investigate a solution for this will make TelegraphCQ a very strong choice concerning home care domain.

Cayuga was not a relevant system for a home care domain, but it will be interesting to compare TelegraphCQ with other system which support real-time analysis of data streams.

References

[CAGP] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, Walker M. White: *Cayuga: A General Purpose Event Monitoring System*. CIDR 2007 412-422

[CEP] http://complexevents.com/

[CHPE] Lars Brenna, Alan J. Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, Walker M. White: *Cayuga: A high-performance event processing engine*. SIGMOD Conference 2007: 1100-1102.

[CMAN] Mingsheng Hong.*Cayuga User Manual.* 2008

[COM] David Luckham. *A short History of Complex Event Processing Part 1: Beginnings.* 2007

[CPS] David C. Luckham and Brian Frasca. *Complex Event ProceSassing in Distributed Systems.* Stanford University Technical Report CSL-TR-98-754, March 1998, 28 pages.

[COM2] David Luckham. *A short History of Complex Event Processing Part 2: the rise of CEP.* 2007

[CWW] http://www.cs.cornell.edu/database/cayuga/

[DSS] Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J. Models and issues in data stream systems; PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. 2002

[DSM] Golab, L., Tamer Ozsu, M. *Issues in data stream management"; SIGMOD Rec., Volume 32, No. 2.* 2003

[EDY] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, Vijayshankar Raman. *Continuously Adaptive Continuous Queries over Streams.* UC Berkeley, IBM Almaden Research Center

[GIGA] Chuck Cranor, Y. Gao, Theodore Johnson, Vladislav Shkapenyuk, and Oliver Spataschek. *Gigascope: High performance network monitoring with an sql interface*. In Proceedings of the 21st ACM SIGMOD International Conference on Management of Data / Principles of Database Systems, June 2002.

[ICI] Lindeberg, M. *Data Stream Management Systems (DSMS) – Introduction, Concepts and Issues*; University of Oslo, INF5100 Advanced Database Systems lecture notes, 10/10/2007

[INF5090] Vera Goebel, Thomas Plagemann *Data Stream Management Systems – Applications, Concepts and Systems*; University of Oslo, INF5190 Advanced Topics in Distribuerte Systems lecture notes, 2008

[MTJ] Jarle Søberg. Implementation,and Evaluation of Network Monitoring Tasks with the TelegraphCQ Data Stream Management System. Master Thesis 2006

[NEXT] Walker White, Mirek Riedewald, Johannes Gehrke, Alan Demers *What is "Next" in Event Processing?*.Cornell University 2006

[TCDP] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Vijayshankar Raman, Fred Reiss, and Mehul A. Shah. *TelegraphCQ: Continuous Dataflow Processing for an Uncertain World*. CIDR 2003

[TEPS] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. *Towards expressive publish/subscribe systems*. In Proc. EDBT, 2006

[TWW] http://telegraph.cs.berkeley.edu/telegraphcq/v2.1/

[UCEP] Jarle Søberg, André Rodrigues, Vera Goebel, and Thomas Plagemann. *Using Complex Event Processing of Data Streams for Movement Tracking in Home Care Environments*

[WSN01] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam and E. Cayirc. *Wireless sensor networks: a survey*. Computer Networks, 2002 Volume 38, Issure 4, 15 March 2002, Pages 393-422.

[QPSN] Y. Yao and J. E. Gehrke. *Query processing for sensor networks*. In Proceedings of the 2003 Conference on Innovative Data Systems Research *(CIDR 2003)*, January 2003.

Appendix

CAYUGA

Stream.txt

A` 20` 1` 1` Sensors` 1` 1
B` 36` 1` 1` Sensors` 1` 1
C` 0` 1` 1` Sensors` 1` 1
N` 1` 1` 1` Sensors` 1` 1
M` 0` 1` 1` Sensors` 1` 1
A` 26` 2` 2` Sensors` 2` 2
B` 19` 2` 2` Sensors` 2` 2
C` 0` 2` 2` Sensors` 2` 2
N` 1` 2` 2` Sensors` 2` 2
M` 0` 2` 2` Sensors` 2` 2
A` 36` 3` 3` Sensors` 3` 3
B` 15` 3` 3` Sensors` 3` 3
C` 1` 3` 3` Sensors` 3` 3
N` 1` 3` 3` Sensors` 3` 3
M` 0` 3` 3` Sensors` 3` 3
A` 35` 4` 4` Sensors` 4` 4
B` 33` 4` 4` Sensors` 4` 4
C` 1` 4` 4` Sensors` 4` 4
N` 0` 4` 4` Sensors` 4` 4
M` 0` 4` 4` Sensors` 4` 4
A` 28` 5` 5` Sensors` 5` 5
B` 8` 5` 5` Sensors` 5` 5
C` 1` 5` 5` Sensors` 5` 5
N` 1` 5` 5` Sensors` 5` 5
M` 0` 5` 5` Sensors` 5` 5
A` 37` 6` 6` Sensors` 6` 6
B` 28` 6` 6` Sensors` 6` 6
C` 0` 6` 6` Sensors` 6` 6
N` 0` 6` 6` Sensors` 6` 6
M` 1` 6` 6` Sensors` 6` 6
A` 17` 7` 7` Sensors` 7` 7
B` 31` 7` 7` Sensors` 7` 7
C` 1` 7` 7` Sensors` 7` 7
N` 0` 7` 7` Sensors` 7` 7
M` 1` 7` 7` Sensors` 7` 7
A` 4` 8` 8` Sensors` 8` 8
B` 28` 8` 8` Sensors` 8` 8

C` 1` 8` 8` Sensors` 8` 8
N` 1` 8` 8` Sensors` 8` 8
M` 0` 8` 8` Sensors` 8` 8
A` 40` 9` 9` Sensors` 9` 9
B` 31` 9` 9` Sensors` 9` 9
C` 1` 9` 9` Sensors` 9` 9
N` 0` 9` 9` Sensors` 9` 9
M` 1` 9` 9` Sensors` 9` 9
A` 12` 10` 10` Sensors` 10` 10
B` 9` 10` 10` Sensors` 10` 10
C` 0` 10` 10` Sensors` 10` 10
N` 0` 10` 10` Sensors` 10` 10
M` 0` 10` 10` Sensors` 10` 10
A` 7` 11` 11` Sensors` 11` 11
B` 37` 11` 11` Sensors` 11` 11
C` 0` 11` 11` Sensors` 11` 11
N` 0` 11` 11` Sensors` 11` 11
M` 1` 11` 11` Sensors` 11` 11
A` 30` 12` 12` Sensors` 12` 12
B` 13` 12` 12` Sensors` 12` 12
C` 1` 12` 12` Sensors` 12` 12
N` 0` 12` 12` Sensors` 12` 12
M` 0` 12` 12` Sensors` 12` 12
A` 19` 13` 13` Sensors` 13` 13
B` 29` 13` 13` Sensors` 13` 13
C` 0` 13` 13` Sensors` 13` 13
N` 0` 13` 13` Sensors` 13` 13
M` 1` 13` 13` Sensors` 13` 13
A` 29` 14` 14` Sensors` 14` 14
B` 15` 14` 14` Sensors` 14` 14
C` 0` 14` 14` Sensors` 14` 14
N` 0` 14` 14` Sensors` 14` 14
M` 1` 14` 14` Sensors` 14` 14
A` 14` 27` 27` Sensors` 27` 27
B` 36` 27` 27` Sensors` 27` 27
C` 1` 27` 27` Sensors` 27` 27
N` 1` 27` 27` Sensors` 27` 27
M` 1` 27` 27` Sensors` 27` 27
A` 23` 28` 29` Sensors` 28` 29
B` 12` 28` 29` Sensors` 28` 29
C` 1` 28` 29` Sensors` 28` 29
N` 0` 28` 29` Sensors` 28` 29
M` 0` 28` 29` Sensors` 28` 29
A` 1` 31` 33` Sensors` 31` 33
B` 14` 31` 33` Sensors` 31` 33
C` 1` 31` 33` Sensors` 31` 33
N` 1` 31` 33` Sensors` 31` 33

M` 0` 31` 33` Sensors` 31` 33
A` 37` 32` 35` Sensors` 32` 35
B` 37` 32` 35` Sensors` 32` 35
C` 1` 32` 35` Sensors` 32` 35
N` 0` 32` 35` Sensors` 32` 35
M` 0` 32` 35` Sensors` 32` 35
A` 16` 36` 40` Sensors` 36` 40
B` 6` 36` 40` Sensors` 36` 40
C` 1` 36` 40` Sensors` 36` 40
N` 0` 36` 40` Sensors` 36` 40
M` 1` 36` 40` Sensors` 36` 40
A` 40` 40` 45` Sensors` 40` 45
B` 4` 40` 45` Sensors` 40` 45
C` 1` 40` 45` Sensors` 40` 45
N` 1` 40` 45` Sensors` 40` 45
M` 0` 40` 45` Sensors` 40` 45
A` 22` 57` 60` Sensors` 57` 60
B` 41` 57` 60` Sensors` 57` 60
C` 1` 57` 60` Sensors` 57` 60
N` 1` 57` 60` Sensors` 57` 60
M` 0` 57` 60` Sensors` 57` 60

SensorsSchema.xml

```
<?xml version="1.0" encoding="utf-8"?>
<StreamType xmlns="http://tempuri.org/SensorsSchema.xsd" Name="Sensors">
    <AttrNameType Name="ID" Type="string"/>
    <AttrNameType Name="Value" Type="int"/>
    <AttrNameType Name="T0" Type="int"/>
    <AttrNameType Name="T1" Type="int"/>
</StreamType>
```

Queries

```
::::::::::::::
cayuga1_1.txt
::::::::::::::
SELECT *, Value AS Asum, 1 AS Acount, Value AS Aavg
FROM FILTER{ID = 'A' AND Value > 0 } Sensors
PUBLISH SensorsA

::::::::::::::
cayuga1_2.txt
::::::::::::::
```

```
SELECT *, Value AS Bsum, 1 AS Bcount, Value AS Bavg
FROM FILTER{ID = 'B' and Value > 0} Sensors
PUBLISH SensorsB
::::::::::::::
cayuga1_3.txt
::::::::::::::
SELECT *, Value
FROM SensorsA NEXT{ } SensorsB
PUBLISH SensorsAB
::::::::::::::
cayuga1_4.txt
::::::::::::::
SELECT *
FROM SensorsAB FOLD{,
                    $1.T0 < $2.T1 + 5,
                    $.Asum + $2.Value AS Asum,
                    $.Bsum + $2.Value AS Bsum,
                    $.Acount + 1 AS Acount,
                    $.Bcount + 1 AS Bcount,
                    ($.Asum + $2.Value)/($.Acount + 1) AS Aavg,
                    ($.Bsum + $2.Value)/($.Bcount + 1) AS Bavg}
SensorsAB
PUBLISH SensorsABagg
::::::::::::::
cayuga1_5.txt
::::::::::::::
SELECT
FROM FILTER{DUR >= 5 AND Aavg > Bavg} SensorsABagg
PUBLISH Query1



::::::::::::::
cayuga2_1.txt
::::::::::::::
SELECT ID, T0, T1, Value
FROM FILTER {ID = 'N' AND Value = 1} Sensors
PUBLISH SensorN
::::::::::::::
cayuga2_2.txt
::::::::::::::
SELECT ID, T0, T1, Value
FROM FILTER {ID = 'C' AND Value = 1 } Sensors
PUBLISH SensorC
::::::::::::::
```

cayuga2_3.txt
::::::::::::::
SELECT T0, T1, Value
FROM SensorN FOLD {, ($1.T0 < $2.T0) as T0 AND ($1.T1 < $2.T1) as T1 AND DUR = 1 ,}
SensorC
PUBLISH SensorNC
::::::::::::::
cayuga2_4.txt
::::::::::::::
SELECT ID, T0, T1, Value
FROM FILTER {ID = 'M' AND Value = 0 } Sensors
PUBLISH SensorM
::::::::::::::
cayuga2_5.txt
::::::::::::::
SELECT T0, T1
FROM SensorNC NEXT {$1.T0 = T0 AND $1.T1 = T1} SensorM
PUBLISH Query2

::::::::::::::
cayuga3_1.txt
::::::::::::::
SELECT *
FROM FILTER {ID = 'M' AND Value = 1} Sensors
PUBLISH SensorM
::::::::::::::
cayuga3_2.txt
::::::::::::::
SELECT *
FROM FILTER {ID = 'N' AND Value = 0 } Sensors
PUBLISH SensorN
::::::::::::::
cayuga3_3.txt
::::::::::::::
SELECT *
FROM SensorM FOLD {, $1.T0 = $.T0 AND $1.T1 = $.T1 AND DUR <= 4 ,} SensorN
PUBLISH Query3

::::::::::::::
cayuga4_1.txt
::::::::::::::

```
SELECT ID, Value, T0, T1
FROM FILTER {ID = 'M' AND (Value < 2) AS Value } Sensors
PUBLISH SensorM

::::::::::::::
cayuga4_2.txt
::::::::::::::
SELECT ID, Value, T0, T1
FROM FILTER {ID = 'N' AND (Value < 2) AS Value } Sensors
PUBLISH SensorN

::::::::::::::
cayuga4_3.txt
::::::::::::::
SELECT *
FROM SensorN FOLD { , $1.T0 = $2.T0 AND $1.T1 = $2.T1  AND $1.Value + $2.Value = 1 , }
SensorM
PUBLISH Query4



::::::::::::::
cayuga5_1.txt
::::::::::::::
SELECT ID, Value, T0, T1
FROM FILTER {ID = 'N' AND Value = 1} Sensors
PUBLISH SensorN

::::::::::::::
cayuga5_2.txt
::::::::::::::
SELECT ID, Value, T0, T1
FROM FILTER {ID = 'M' AND Value = 1} Sensors
PUBLISH SensorM
::::::::::::::
cayuga5_3.txt
::::::::::::::
SELECT *
FROM SensorN FOLD {, $1.T0 = T0 AND $1.T1 = T1 AND DUR = 1 , } SensorM
PUBLISH Query5
```

Config files

```
<?xml version="1.0" encoding="utf-8"?>
<Config xmlns="http://tempuri.org/ConfigSchema.xsd">
  <Option Name="QueryInputMode" Value="FILE"/>
  <Option Name="AirQuery" Value="false"/>
  <Option Name="QueryInputName"
Value="req3/queries/cayuga1_1.txt;req3/queries/cayuga1_2.txt;req3/queries/cayuga1_
3.txt;req3/queries/cayuga1_4.txt;req3/quer
ies/cayuga1_5.txt"/>
  <Option Name="QueryNumber" Value="5"/>
  <Option Name="StreamSchema" Value="req3/schemas/SensorsSchema.xml"/>
  <Option Name="DocInputMode" Value="FILE"/>
  <Option Name="DocInputName" Value="req3/streams/Stream.txt"/>
  <Option Name="DocInputStream" Value="Sensors"/>
  <Option Name="DocNumber" Value="1"/>
  <Option Name="Verbose" Value="true"/>
  <Option Name="RecordTrace" Value="true"/>
  <Option Name="Measure" Value="true"/>
  <Option Name="CheckPointFrequency" Value="1"/>
  <Option Name="CheckPointAndTraceDir" Value="log/"/>
  <Option Name="AttrDelimiter" Value="`"/>
</Config>
```

```
<?xml version="1.0" encoding="utf-8"?>
<Config xmlns="http://tempuri.org/ConfigSchema.xsd">
  <Option Name="QueryInputMode" Value="FILE"/>
  <Option Name="AirQuery" Value="false"/>
  <Option Name="QueryInputName"
Value="req3/queries/cayuga2_1.txt;req3/queries/cayuga2_2.txt;req3/queries/cayuga2_
3.txt;req3/queries/cayuga2_4.txt;req3/quer
ies/cayuga2_5.txt"/>
  <Option Name="QueryNumber" Value="5"/>
  <Option Name="StreamSchema" Value="req3/schemas/SensorsSchema.xml"/>
  <Option Name="DocInputMode" Value="FILE"/>
  <Option Name="DocInputName" Value="req3/streams/Stream.txt"/>
  <Option Name="DocInputStream" Value="Sensors"/>
  <Option Name="DocNumber" Value="1"/>
  <Option Name="Verbose" Value="true"/>
  <Option Name="RecordTrace" Value="true"/>
  <Option Name="Measure" Value="true"/>
  <Option Name="CheckPointFrequency" Value="1"/>
  <Option Name="CheckPointAndTraceDir" Value="log/"/>
  <Option Name="AttrDelimiter" Value="`"/>
</Config>
```

```
<?xml version="1.0" encoding="utf-8"?>
<Config xmlns="http://tempuri.org/ConfigSchema.xsd">
 <Option Name="QueryInputMode" Value="FILE"/>
 <Option Name="AirQuery" Value="false"/>
 <Option Name="QueryInputName"
Value="req3/queries/cayuga3_1.txt;req3/queries/cayuga3_2.txt;req3/queries/cayuga3_
3.txt"/>
 <Option Name="QueryNumber" Value="3"/>
 <Option Name="StreamSchema" Value="req3/schemas/SensorsSchema.xml"/>
 <Option Name="DocInputMode" Value="FILE"/>
 <Option Name="DocInputName" Value="req3/streams/Stream.txt"/>
 <Option Name="DocInputStream" Value="Sensors"/>
 <Option Name="DocNumber" Value="1"/>
 <Option Name="Verbose" Value="true"/>
 <Option Name="RecordTrace" Value="true"/>
 <Option Name="Measure" Value="true"/>
 <Option Name="CheckPointFrequency" Value="1"/>
 <Option Name="CheckPointAndTraceDir" Value="log/"/>
 <Option Name="AttrDelimiter" Value="`"/>
</Config>
```

```
<?xml version="1.0" encoding="utf-8"?>
<Config xmlns="http://tempuri.org/ConfigSchema.xsd">
 <Option Name="QueryInputMode" Value="FILE"/>
 <Option Name="AirQuery" Value="false"/>
 <Option Name="QueryInputName"
Value="req3/queries/cayuga4_1.txt;req3/queries/cayuga4_2.txt;req3/queries/cayuga4_
3.txt"/>
 <Option Name="QueryNumber" Value="3"/>
 <Option Name="StreamSchema" Value="req3/schemas/SensorsSchema.xml"/>
 <Option Name="DocInputMode" Value="FILE"/>
 <Option Name="DocInputName" Value="req3/streams/Stream.txt"/>
 <Option Name="DocInputStream" Value="Sensors"/>
 <Option Name="DocNumber" Value="1"/>
 <Option Name="Verbose" Value="true"/>
 <Option Name="RecordTrace" Value="true"/>
 <Option Name="Measure" Value="true"/>
 <Option Name="CheckPointFrequency" Value="1"/>
 <Option Name="CheckPointAndTraceDir" Value="log/"/>
 <Option Name="AttrDelimiter" Value="`"/>
</Config>
```

```
<?xml version="1.0" encoding="utf-8"?>
<Config xmlns="http://tempuri.org/ConfigSchema.xsd">
 <Option Name="QueryInputMode" Value="FILE"/>
 <Option Name="AirQuery" Value="false"/>
 <Option Name="QueryInputName"
Value="req3/queries/cayuga5_1.txt;req3/queries/cayuga5_2.txt;req3/queries/cayuga5_
3.txt"/>
 <Option Name="QueryNumber" Value="3"/>
 <Option Name="StreamSchema" Value="req3/schemas/SensorsSchema.xml"/>
 <Option Name="DocInputMode" Value="FILE"/>
 <Option Name="DocInputName" Value="req3/streams/Stream.txt"/>
 <Option Name="DocInputStream" Value="Sensors"/>
 <Option Name="DocNumber" Value="1"/>
 <Option Name="Verbose" Value="true"/>
 <Option Name="RecordTrace" Value="true"/>
 <Option Name="Measure" Value="true"/>
 <Option Name="CheckPointFrequency" Value="1"/>
 <Option Name="CheckPointAndTraceDir" Value="log/"/>
 <Option Name="AttrDelimiter" Value="`"/>
</Config>
```

Witnesses.txt

```
::::::::::::::
Task 1
::::::::::::::

  -

::::::::::::::
Task 2
::::::::::::::
12`WITNESS`ID`N`T0`1`T1`1`Value`1`SensorN`1`1`
12`WITNESS`ID`M`T0`1`T1`1`Value`0`SensorM`1`1`
12`WITNESS`ID`N`T0`2`T1`2`Value`1`SensorN`2`2`
12`WITNESS`ID`M`T0`2`T1`2`Value`0`SensorM`2`2`
12`WITNESS`ID`C`T0`3`T1`3`Value`1`SensorC`3`3`
12`WITNESS`ID`N`T0`3`T1`3`Value`1`SensorN`3`3`
12`WITNESS`ID`M`T0`3`T1`3`Value`0`SensorM`3`3`
10`WITNESS`T0`3`T1`3`Value`1`SensorNC`2`3`
12`WITNESS`ID`C`T0`4`T1`4`Value`1`SensorC`4`4`
12`WITNESS`ID`M`T0`4`T1`4`Value`0`SensorM`4`4`
```

10`WITNESS`T0`4`T1`4`Value`1`SensorNC`3`4`
8`WITNESS`T0`4`T1`4`Query2`2`4`
12`WITNESS`ID`C`T0`5`T1`5`Value`1`SensorC`5`5`
12`WITNESS`ID`N`T0`5`T1`5`Value`1`SensorN`5`5`
12`WITNESS`ID`M`T0`5`T1`5`Value`0`SensorM`5`5`
8`WITNESS`T0`5`T1`5`Query2`3`5`
12`WITNESS`ID`C`T0`7`T1`7`Value`1`SensorC`7`7`
12`WITNESS`ID`C`T0`8`T1`8`Value`1`SensorC`8`8`
12`WITNESS`ID`N`T0`8`T1`8`Value`1`SensorN`8`8`
12`WITNESS`ID`M`T0`8`T1`8`Value`0`SensorM`8`8`
12`WITNESS`ID`C`T0`9`T1`9`Value`1`SensorC`9`9`
10`WITNESS`T0`9`T1`9`Value`1`SensorNC`8`9`
12`WITNESS`ID`M`T0`10`T1`10`Value`0`SensorM`10`10`
8`WITNESS`T0`10`T1`10`Query2`8`10`
12`WITNESS`ID`C`T0`12`T1`12`Value`1`SensorC`12`12`
12`WITNESS`ID`M`T0`12`T1`12`Value`0`SensorM`12`12`
12`WITNESS`ID`C`T0`27`T1`27`Value`1`SensorC`27`27`
12`WITNESS`ID`N`T0`27`T1`27`Value`1`SensorN`27`27`
12`WITNESS`ID`C`T0`28`T1`29`Value`1`SensorC`28`29`
12`WITNESS`ID`M`T0`28`T1`29`Value`0`SensorM`28`29`
12`WITNESS`ID`C`T0`31`T1`33`Value`1`SensorC`31`33`
12`WITNESS`ID`N`T0`31`T1`33`Value`1`SensorN`31`33`
12`WITNESS`ID`M`T0`31`T1`33`Value`0`SensorM`31`33`
12`WITNESS`ID`C`T0`32`T1`35`Value`1`SensorC`32`35`
12`WITNESS`ID`M`T0`32`T1`35`Value`0`SensorM`32`35`
12`WITNESS`ID`C`T0`36`T1`40`Value`1`SensorC`36`40`
12`WITNESS`ID`C`T0`40`T1`45`Value`1`SensorC`40`45`
12`WITNESS`ID`N`T0`40`T1`45`Value`1`SensorN`40`45`
12`WITNESS`ID`M`T0`40`T1`45`Value`0`SensorM`40`45`
12`WITNESS`ID`C`T0`57`T1`60`Value`1`SensorC`57`60`
12`WITNESS`ID`N`T0`57`T1`60`Value`1`SensorN`57`60`
12`WITNESS`ID`M`T0`57`T1`60`Value`0`SensorM`57`60`

::::::::::::::
Task 3
::::::::::::::
12`WITNESS`ID`N`Value`0`T0`4`T1`4`SensorN`4`4`
12`WITNESS`ID`N`Value`0`T0`6`T1`6`SensorN`6`6`
12`WITNESS`ID`M`Value`1`T0`6`T1`6`SensorM`6`6`
12`WITNESS`ID`N`Value`0`T0`7`T1`7`SensorN`7`7`
12`WITNESS`ID`M`Value`1`T0`7`T1`7`SensorM`7`7`
20`WITNESS`ID_1`M`Value_1`1`T0_1`6`T1_1`6`ID`N`Value`0`T0`7`T1`7`Query3`6`7`
12`WITNESS`ID`N`Value`0`T0`9`T1`9`SensorN`9`9`
12`WITNESS`ID`M`Value`1`T0`9`T1`9`SensorM`9`9`

20`WITNESS`ID_1`M`Value_1`1`T0_1`7`T1_1`7`ID`N`Value`0`T0`9`T1`9`Query3`7`9`
12`WITNESS`ID`N`Value`0`T0`10`T1`10`SensorN`10`10`
20`WITNESS`ID_1`M`Value_1`1`T0_1`9`T1_1`9`ID`N`Value`0`T0`10`T1`10`Query3`9`10`
12`WITNESS`ID`N`Value`0`T0`11`T1`11`SensorN`11`11`
12`WITNESS`ID`M`Value`1`T0`11`T1`11`SensorM`11`11`
12`WITNESS`ID`N`Value`0`T0`12`T1`12`SensorN`12`12`
20`WITNESS`ID_1`M`Value_1`1`T0_1`11`T1_1`11`ID`N`Value`0`T0`12`T1`12`Query3`11`12`
12`WITNESS`ID`N`Value`0`T0`13`T1`13`SensorN`13`13`
12`WITNESS`ID`M`Value`1`T0`13`T1`13`SensorM`13`13`
12`WITNESS`ID`N`Value`0`T0`14`T1`14`SensorN`14`14`
12`WITNESS`ID`M`Value`1`T0`14`T1`14`SensorM`14`14`
20`WITNESS`ID_1`M`Value_1`1`T0_1`13`T1_1`13`ID`N`Value`0`T0`14`T1`14`Query3`13`14`
12`WITNESS`ID`M`Value`1`T0`27`T1`27`SensorM`27`27`
12`WITNESS`ID`N`Value`0`T0`28`T1`29`SensorN`28`29`
20`WITNESS`ID_1`M`Value_1`1`T0_1`27`T1_1`27`ID`N`Value`0`T0`28`T1`29`Query3`27`29`
12`WITNESS`ID`N`Value`0`T0`32`T1`35`SensorN`32`35`
12`WITNESS`ID`N`Value`0`T0`36`T1`40`SensorN`36`40`
12`WITNESS`ID`M`Value`1`T0`36`T1`40`SensorM`36`40`

::::::::::::::
Task 4
::::::::::::::

12`WITNESS`ID`N`Value`1`T0`1`T1`1`SensorN`1`1`
12`WITNESS`ID`M`Value`0`T0`1`T1`1`SensorM`1`1`
12`WITNESS`ID`N`Value`1`T0`2`T1`2`SensorN`2`2`
12`WITNESS`ID`M`Value`0`T0`2`T1`2`SensorM`2`2`
12`WITNESS`ID`N`Value`1`T0`3`T1`3`SensorN`3`3`
12`WITNESS`ID`M`Value`0`T0`3`T1`3`SensorM`3`3`
12`WITNESS`ID`N`Value`0`T0`4`T1`4`SensorN`4`4`
12`WITNESS`ID`M`Value`0`T0`4`T1`4`SensorM`4`4`
12`WITNESS`ID`N`Value`1`T0`5`T1`5`SensorN`5`5`
12`WITNESS`ID`M`Value`0`T0`5`T1`5`SensorM`5`5`
12`WITNESS`ID`N`Value`0`T0`6`T1`6`SensorN`6`6`
12`WITNESS`ID`M`Value`1`T0`6`T1`6`SensorM`6`6`
12`WITNESS`ID`N`Value`0`T0`7`T1`7`SensorN`7`7`
12`WITNESS`ID`M`Value`1`T0`7`T1`7`SensorM`7`7`
12`WITNESS`ID`N`Value`1`T0`8`T1`8`SensorN`8`8`
12`WITNESS`ID`M`Value`0`T0`8`T1`8`SensorM`8`8`
12`WITNESS`ID`N`Value`0`T0`9`T1`9`SensorN`9`9`

12`WITNESS`ID`M`Value`1`T0`9`T1`9`SensorM`9`9`
12`WITNESS`ID`N`Value`0`T0`10`T1`10`SensorN`10`10`
12`WITNESS`ID`M`Value`0`T0`10`T1`10`SensorM`10`10`
12`WITNESS`ID`N`Value`0`T0`11`T1`11`SensorN`11`11`
12`WITNESS`ID`M`Value`1`T0`11`T1`11`SensorM`11`11`
12`WITNESS`ID`N`Value`0`T0`12`T1`12`SensorN`12`12`
12`WITNESS`ID`M`Value`0`T0`12`T1`12`SensorM`12`12`
12`WITNESS`ID`N`Value`0`T0`13`T1`13`SensorN`13`13`
12`WITNESS`ID`M`Value`1`T0`13`T1`13`SensorM`13`13`
12`WITNESS`ID`N`Value`0`T0`14`T1`14`SensorN`14`14`
12`WITNESS`ID`M`Value`1`T0`14`T1`14`SensorM`14`14`
12`WITNESS`ID`N`Value`1`T0`27`T1`27`SensorN`27`27`
12`WITNESS`ID`M`Value`1`T0`27`T1`27`SensorM`27`27`
12`WITNESS`ID`N`Value`0`T0`28`T1`29`SensorN`28`29`
12`WITNESS`ID`M`Value`0`T0`28`T1`29`SensorM`28`29`
12`WITNESS`ID`N`Value`1`T0`31`T1`33`SensorN`31`33`
12`WITNESS`ID`M`Value`0`T0`31`T1`33`SensorM`31`33`
12`WITNESS`ID`N`Value`0`T0`32`T1`35`SensorN`32`35`
12`WITNESS`ID`M`Value`0`T0`32`T1`35`SensorM`32`35`
12`WITNESS`ID`N`Value`0`T0`36`T1`40`SensorN`36`40`
12`WITNESS`ID`M`Value`1`T0`36`T1`40`SensorM`36`40`
12`WITNESS`ID`N`Value`1`T0`40`T1`45`SensorN`40`45`
12`WITNESS`ID`M`Value`0`T0`40`T1`45`SensorM`40`45`
12`WITNESS`ID`N`Value`1`T0`57`T1`60`SensorN`57`60`
12`WITNESS`ID`M`Value`0`T0`57`T1`60`SensorM`57`60`

::::::::::::::
Task 5
::::::::::::::

12`WITNESS`ID`N`Value`1`T0`1`T1`1`SensorN`1`1`
12`WITNESS`ID`N`Value`1`T0`2`T1`2`SensorN`2`2`
12`WITNESS`ID`N`Value`1`T0`3`T1`3`SensorN`3`3`
12`WITNESS`ID`N`Value`1`T0`5`T1`5`SensorN`5`5`
12`WITNESS`ID`M`Value`1`T0`6`T1`6`SensorM`6`6`
20`WITNESS`ID_1`N`Value_1`1`T0_1`5`T1_1`5`ID`M`Value`1`T0`6`T1`6`Query5`5`6`
12`WITNESS`ID`M`Value`1`T0`7`T1`7`SensorM`7`7`
12`WITNESS`ID`N`Value`1`T0`8`T1`8`SensorN`8`8`
12`WITNESS`ID`M`Value`1`T0`9`T1`9`SensorM`9`9`
20`WITNESS`ID_1`N`Value_1`1`T0_1`8`T1_1`8`ID`M`Value`1`T0`9`T1`9`Query5`8`9`
12`WITNESS`ID`M`Value`1`T0`11`T1`11`SensorM`11`11`
12`WITNESS`ID`M`Value`1`T0`13`T1`13`SensorM`13`13`
12`WITNESS`ID`M`Value`1`T0`14`T1`14`SensorM`14`14`

12`WITNESS`ID`N`Value`1`T0`27`T1`27`SensorN`27`27`
12`WITNESS`ID`M`Value`1`T0`27`T1`27`SensorM`27`27`
12`WITNESS`ID`N`Value`1`T0`31`T1`33`SensorN`31`33`
12`WITNESS`ID`M`Value`1`T0`36`T1`40`SensorM`36`40`
12`WITNESS`ID`N`Value`1`T0`40`T1`45`SensorN`40`45`
12`WITNESS`ID`N`Value`1`T0`57`T1`60`SensorN`57`60`

TELEGRAPHCQ

Stream.txt

A, 20, 1, 1
B, 36, 1, 1
C, 0, 1, 1
N, 1, 1, 1
M, 0, 1, 1
A, 26, 2, 2
B, 19, 2, 2
C, 0, 2, 2
N, 1, 2, 2
M, 0, 2, 2
A, 36, 3, 3
B, 15, 3, 3
C, 1, 3, 3
N, 1, 3, 3
M, 0, 3, 3
A, 35, 4, 4
B, 33, 4, 4
C, 1, 4, 4
N, 0, 4, 4
M, 0, 4, 4
A, 28, 5, 5
B, 8, 5, 5
C, 1, 5, 5
N, 1, 5, 5
M, 0, 5, 5
A, 37, 6, 6
B, 28, 6, 6
C, 0, 6, 6
N, 0, 6, 6
M, 1, 6, 6
A, 17, 7, 7
B, 31, 7, 7

C, 1, 7, 7
N, 0, 7, 7
M, 1, 7, 7
A, 4, 8, 8
B, 28, 8, 8
C, 1, 8, 8
N, 1, 8, 8
M, 0, 8, 8
A, 40, 9, 9
B, 31, 9, 9
C, 1, 9, 9
N, 0, 9, 9
M, 1, 9, 9
A, 12, 10, 10
B, 9, 10, 10
C, 0, 10, 10
N, 0, 10, 10
M, 0, 10, 10
A, 7, 11, 11
B, 37, 11, 11
C, 0, 11, 11
N, 0, 11, 11
M, 1, 11, 11
A, 30, 12, 12
B, 13, 12, 12
C, 1, 12, 12
N, 0, 12, 12
M, 0, 12, 12
A, 19, 13, 13
B, 29, 13, 13
C, 0, 13, 13
N, 0, 13, 13
M, 1, 13, 13
A, 29, 14, 14
B, 15, 14, 14
C, 0, 14, 14
N, 0, 14, 14
M, 1, 14, 14
A, 14, 27, 27
B, 36, 27, 27
C, 1, 27, 27
N, 1, 27, 27
M, 1, 27, 27
A, 23, 28, 29
B, 12, 28, 29

```
C, 1, 28, 29
N, 0, 28, 29
M, 0, 28, 29
A, 1, 31, 33,
B, 14, 31, 33
C, 1, 31, 33
N, 1, 31, 33
M, 0, 31, 33
A, 37, 32, 35
B, 37, 32, 35
C, 1, 32, 35
N, 0, 32, 35
M, 0, 32, 35
A, 16, 36, 40
B, 6, 36, 40
C, 1, 36, 40
N, 0, 36, 40
M, 1, 36, 40
A, 40, 40, 45
B, 4, 40, 45
C, 1, 40, 45
N, 1, 40, 45
M, 0, 40, 45
A, 22, 57, 60
B, 41, 57, 60
C, 1, 57, 60
N, 1, 57, 60
M, 0, 57, 60
```

Schema.sql

```
drop schema streams;
create schema streams;

drop stream streams.cep;

create stream streams.cep (
    ID text,
    value int,
    t0 int,
    t1 int,
    tcqtime timestamp TIMESTAMPCOLUMN
) type unarchived;
```

```
alter stream streams.cep add wrapper csvwrapper;
```

Queries

```
::::::::::::::::::::::::::::
Telegraphcq1.sql
::::::::::::::::::::::::::::
drop stream streams.avgA;

create stream streams.avgA (
      ID text,
      averageA float,
      tcqtime timestamp TIMESTAMPCOLUMN
) type unarchived;

alter stream streams.avgA add wrapper csvwrapper;

drop stream streams.avgB;

create stream streams.avgB (
      ID text,
      averageB float,
      tcqtime timestamp TIMESTAMPCOLUMN
) type unarchived;

alter stream streams.avgB add wrapper csvwrapper;

WITH
      streams.avgA
      AS
      (SELECT ID, AVG(value) as averageA, wtime(*)
      FROM streams.cep [RANGE BY '2 seconds' SLIDE BY '1 seconds']
      WHERE ID = 'A'
      GROUP BY ID)

      streams.avgB
      AS
      (SELECT ID, AVG(value) as avgerageB, wtime(*)
      FROM streams.cep [RANGE BY '2 seconds' SLIDE BY '1 seconds']
      WHERE ID = 'B'
      GROUP BY ID)
```

```
(SELECT A.ID, A.averageA, B.ID, B.averageB, A.tcqtime
FROM streams.avgA AS A,
    streams.avgB AS B
WHERE A.tcqtime = B.tcqtime AND A.averageA > B.averageB);



::::::::::::::
telegraph2.sql
::::::::::::::
drop stream streams.readN;

create stream streams.readN (
      ID text,
      value int,
      t0 int,
      tcqtime timestamp TIMESTAMPCOLUMN
) type unarchived;

alter stream streams.readN add wrapper csvwrapper;

drop stream streams.readC;

create stream streams.readC (
      ID text,
      t0 int,
      value int,
      tcqtime timestamp TIMESTAMPCOLUMN
) type unarchived;

alter stream streams.readC add wrapper csvwrapper;

drop stream streams.readM;

create stream streams.readM (
      ID text,
      t0 int,
      value int,
      tcqtime timestamp TIMESTAMPCOLUMN
) type unarchived;

alter stream streams.readM add wrapper csvwrapper;


WITH
```

```
	streams.readN
	AS
	(SELECT ID, t0, value, wtime(*)
	FROM streams.cep [RANGE BY '1 seconds' SLIDE BY '1 seconds']
	WHERE ID = 'N' AND value = '1'
	GROUP BY ID, t0, value)


	streams.readC
	AS
	(SELECT ID, t0, value, wtime(*)
	FROM streams.cep [RANGE BY '1 seconds' SLIDE BY '1 seconds']
	WHERE ID = 'C' AND value = '1'
	GROUP BY ID, t0, value)

	streams.readM
	AS
	(SELECT ID, t0, value, wtime(*)
	FROM streams.cep [RANGE BY '1 seconds' SLIDE BY '1 seconds']
	WHERE ID = 'M' AND value = '0'
	GROUP BY ID, t0, value)

(SELECT C.ID, C.value,C.t0, N.ID, N.value, N.t0, M.ID, M.value, M.t0
FROM  streams.readN AS N,
    streams.readC AS C,
    streams.readM AS M
WHERE N.value = C.value AND N.t0 < C.t0 AND N.t0 = M.t0 );



::::::::::::::
telegraph3.sql
::::::::::::::
drop stream streams.repM;

create stream streams.repM (
	ID text,
	value int,
	t0 int,
	tcqtime timestamp TIMESTAMPCOLUMN
) type unarchived;

alter stream streams.repM add wrapper csvwrapper;

drop stream streams.repN;
```

```
create stream streams.repN (
	ID text,
	value int,
	t0 int,
	tcqtime timestamp TIMESTAMPCOLUMN
) type unarchived;

alter stream streams.repN add wrapper csvwrapper;

WITH

	streams.repM
	AS
	(SELECT ID, value, t0, wtime(*)
	FROM streams.cep [RANGE BY '1 seconds' SLIDE BY '1 seconds']
	WHERE ID = 'M' AND value = '1'
	GROUP BY ID, value, t0)

	streams.repN
	AS
	(SELECT ID, value, t0, wtime(*)
	FROM streams.cep [RANGE BY '1 seconds' SLIDE BY '1 seconds']
	WHERE ID = 'N' AND value = '0'
	GROUP BY ID, value, t0)

(SELECT M.ID, M.value, M.t0, N.ID, N.value, N.t0
FROM streams.repM AS M,
   streams.repN AS N
WHERE M.t0 + 1 = N.t0 AND N.value < M.value);



::::::::::::::
telegraph4.sql
::::::::::::::
drop stream streams.onlyN;

create stream streams.onlyN (
	ID text,
	t0 int,
	value int,
	tcqtime timestamp TIMESTAMPCOLUMN
) type unarchived;

alter stream streams.onlyN add wrapper csvwrapper;
```

```
drop stream streams.onlyM;

create stream streams.onlyM (
      ID text,
      t0 int,
      value int,
      tcqtime timestamp TIMESTAMPCOLUMN
) type unarchived;

alter stream streams.onlyM add wrapper csvwrapper;

WITH
            streams.onlyN
            AS
            (SELECT ID, t0, value, wtime(*)
            FROM streams.cep [RANGE BY '1 seconds' SLIDE BY '1 seconds']
            WHERE ID = 'N'
            GROUP BY ID, t0, value)

            streams.onlyM
            AS
            (SELECT ID, t0, value, wtime(*)
            FROM streams.cep [RANGE BY '1 seconds' SLIDE BY '1 seconds']
            WHERE ID = 'M'
            GROUP BY ID, t0, value)

(SELECT N.ID, N.value, N.t0, M.ID, M.value, M.t0
FROM streams.onlyN AS N,
   streams.onlyM AS M
WHERE N.t0 = M.t0 AND M.value <> N.value);



::::::::::::::
telegraph5.sql
::::::::::::::
drop stream streams.moveN;

create stream streams.moveN (
      ID text,
      t0 int,
      value int,
      tcqtime timestamp TIMESTAMPCOLUMN
) type unarchived;
```

```
alter stream streams.moveN add wrapper csvwrapper;

drop stream streams.moveM;

create stream streams.moveM (
      ID text,
      t0 int,
      value int,
      tcqtime timestamp TIMESTAMPCOLUMN
) type unarchived;

alter stream streams.moveM add wrapper csvwrapper;

WITH
streams.moveN
            AS
            (SELECT ID, t0, value, wtime(*)
            FROM streams.cep [RANGE BY '1 seconds' SLIDE BY '1 seconds']
            WHERE ID = 'N' AND value = '1'
            GROUP BY ID, t0, value)

            streams.moveM
            AS
            (SELECT ID, t0, value, wtime(*)
            FROM streams.cep [RANGE BY '1 seconds' SLIDE BY '1 seconds']
            WHERE ID = 'M' AND value = '1'
            GROUP BY ID, t0, value)

(SELECT N.ID,N.value, N.t0, M.ID, M.value, M.t0
FROM streams.moveN AS N,
   streams.moveM AS M
WHERE N.t0 = M.t0);
```

Result.txt

```
::::::::::::::
resultat1.res
::::::::::::::
A,31,B,17,2009-01-12 11:03:26
A,35.5,B,24,2009-01-12 11:03:27
A,31.5,B,20.5,2009-01-12 11:03:28
A,32.5,B,18,2009-01-12 11:03:29
A,26,B,20,2009-01-12 11:03:33
```

A,24.5,B,21,2009-01-12 11:03:36
A,24,B,22,2009-01-12 11:03:37
A,26.5,B,21.5,2009-01-12 11:03:42
A,28,B,5,2009-01-12 11:03:43
A,31,B,22.5,2009-01-12 11:03:44
success=0, cqcancel=1

::::::::::::::
resultat2.res
::::::::::::::
success=0, cqcancel=1

::::::::::::::
resultat3.res
::::::::::::::
M,1,6,N,0,7
M,1,9,N,0,10
M,1,11,N,0,12
M,1,13,N,0,14
M,1,31,N,0,32
success=0, cqcancel=1

::::::::::::::
resultat4.res
::::::::::::::
N,1,1,M,0,1
N,1,2,M,0,2
N,1,3,M,0,3
N,1,5,M,0,5
N,0,6,M,1,6
N,0,7,M,1,7
N,1,8,M,0,8
N,0,9,M,1,9
N,0,11,M,1,11
N,0,13,M,1,13
N,0,14,M,1,14
N,1,27,M,0,27
N,0,36,M,1,36
N,1,40,M,0,40
success=0, cqcancel=1

::::::::::::::
resultat5.res
::::::::::::::
N,1,31,M,1,31
success=0, cqcancel=1