Chapter 5
Data Streams and Data Stream Management Systems and Languages

Emanuele Panigati, Fabio A. Schreiber, and Carlo Zaniolo

5.1 A Short Introduction to Information Flow Processing and Data Streams

The massive usage of data streams dates back to artificial satellite information processing systems and to their commercial application in the early 1970s, such as in telecommunications switching, land monitoring, meteorological surveillance, etc. Today they are extensively used in monitoring systems applications based on wired and wireless sensor networks, in social networks, and in the Internet of Things [20]. The main functional goals of data stream management systems (DSMSs) are as follows: (a) results must be pushed to the output promptly and eagerly while input tuples continue to arrive and (b) because of the unbounded and massive nature of data streams, all past tuples cannot be memorized for future use. Only synopses can be kept in memory and the rest must be discarded. The main problem created by these goals is a further significant loss of expressive power with respect to the well-known limitations of structured query language (SQL) and other relational languages; unfortunately, these limitations are dramatically more disruptive to data stream applications for the following reasons: (1) blocking query operators that were widely used on databases can no longer be allowed on data streams; (2) database libraries of external functions written in procedural languages are much less effective in DSMSs, which process data in small increments rather than as aggregate large objects; and (3) embedding queries in a procedural programming

E. Panigati • F.A. Schreiber (✉)
DEIB – Politecnico di Milano, Piazza L. da Vinci 32, 20133 Milan, Italy
e-mail: emanuele.panigati@polimi.it; fabio.schreiber@polimi.it

C. Zaniolo
UCLA Computer Science Department, 3532D Boelter Hall, Los Angeles, CA 90095-1596, USA
e-mail: zaniolo@cs.ucla.edu

© Springer International Publishing Switzerland 2015

F. Colace et al. (eds.), *Data Management in Pervasive Systems*, Data-Centric Systems and Applications, DOI 10.1007/978-3-319-20062-0_5

language, a solution used extensively in relational database applications, is of very limited effectiveness on data streams. A key research issue for DSMSs is deciding which data model and query language should be used; a wide spectrum of different solutions have in fact been proposed, including operator-based graphical interfaces, programming language extensions, and an assortment of other solutions provided in publish/subscribe systems. In this chapter, the main features and architectural issues will be examined for DSMSs as an introduction to the following chapters, which provide a detailed account of their usage in different application areas. In particular, we shall introduce several *stream query languages* and *data stream management systems* (DSMSs) developed in different projects and contexts.

5.2 Data Stream Definitions

In this section, we will provide some definitions and general concepts about data streams and their handling, allowing the reader to completely and easily understand the peculiarity of each of the languages and systems presented in Sect. 5.4. Without loss of generality, we shall mostly refer to relational data streams, i.e., streams whose data elements are constituted by relational tuples over a schema $\Re$.

Table 5.1 summarizes some of the most noticeable differences between database management systems (DBMSs) and DSMSs as far as their functional features are concerned.

First of all, let us discuss some peculiar differences between a DBMS and a DSMS. A sort of antisymmetry can be noticed between the data management functions in the two of them: in the database, data are stored in a permanent way, while queries are dynamically created and processed; however, in data streams, queries are permanently stored in the sensors and continuously produce a data stream flowing from the real world to a final processing/storage station. Therefore, DSMS computation is generally *data driven*, computing the results as new data come into the system (or at least periodically), since queries are *continuous queries*,

Table 5.1 Comparison between DBMS and DSMS functional features

Feature	DBMS	DSMS
Model	*Persistent* data	*Transient* data
Table	*Set or bag* of tuples	*Infinite sequence* of tuples
Updates	All	Append only
Queries	Transient	Persistent
Query answers	Exact	Often approximate
Query evaluation	Multi-pass	One pass
Operators	Blocking and non-blocking	Non-blocking
Query plan	Fixed	Adaptive
Data processing	Synchronous	Asynchronous
Concurrency overhead	High	Low

which are issued only once and permanently remain active in the system; the DBMS approach is *query driven*, and computation runs when a specific query is executed (one-time queries). Therefore, the first problem a DSMS must deal with is managing changes in data arrival rate during a specific query lifetime.

Another problem is related to the unboundedness of the stream itself that continuously flows into the system and cannot be stored in a traditional way; therefore, only the most recent data should be used for answering the queries at a given time point.

In order to make these concepts more precise, let us introduce some formal definitions:

- A *stream* S is a countably infinite set of elements $s \in S$.
- A *stream element* is a four-tuple $s: < v, t^{\text{app}}, t^{\text{sys}}, bid >$ where
  - $v \in \Re$ is a relational tuple
  - $t^{\text{app}} \in T$ is a partially ordered application time value in the time domain T
  - $t^{\text{sys}} \in T$ is a totally ordered system time value in T
  - $bid \in \mathbf{N}$ is a batch-id value, where a *batch* B is a finite subset of S where all $b \in S$ have an identical t^{app}

A fundamental concept in data stream systems is that of *window*: due to the unboundedness of the stream, it is necessary to define some slices on it—called windows—in order to allow correct processing in finite time. The window *size* can be defined as a fixed number of data items—also called *snapshots*—or as a fixed time interval, no matter how many data items are included in it. In the latter case, also the *slide*, i.e., the time distance between two consecutive windows, and the possibility of windows overlapping, as in the case of sliding windows shown in Fig. 5.1, must be considered.

A *time-based window* W *=(o,c]* over S is a finite subset of stream elements s where $o < s.t^{\text{app}} \leq c$:

- *Window size* ω: $\forall\ W = (o, c] \in W \Rightarrow (c - o) = \omega$
- *Window slide* β: $\exists\ W_1, W_2 \Rightarrow (o_2 - o_1) = \beta$

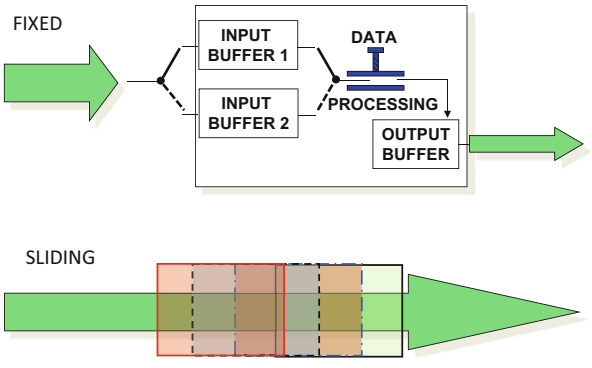

Fig. 5.1 Example of fixed and sliding windows

If $\beta \geq \omega$, windows are disjoint; if $\beta < \omega$, windows are (partially) overlapped.

More definitions and some examples on the usage of windows in the continuous query language (CQL) [3] will be introduced in Chap. 7.

Systems and applications that handle data streams must allow to filter the stream, to compute aggregates, to compose several streams in a unique stream (or decompose a unique stream to several streams), and to find frequent data in the stream and perform some online mining on it.

A stream could be processed in two main ways, *forward* and *backward*, depending on the processing engine implementation, the query language semantics, and the initial processing time point. Both these processing paradigms present problems in handling infinite streams:

Backward: The system may have started long before the moment at which a given query/rule processing has to be evaluated; therefore, lots of data should be processed, causing a heavy workload and very long response time. The solution to the backward infinity-looking problem is to set a backward-looking limit, thus avoiding the system from searching for solutions from very old data and focusing only on the top, most recent ones.

Forward: *Monotonic* operators, such as COUNT and SUM, are *non-blocking* since they can work on data as soon as they arrive; however, some operators, called *blocking operators* (e.g., the `NOT` operator), require that a single query/rule has to process a finite set of data which must be known when the computation starts; otherwise their execution may be delayed infinitely, causing processes to starve. The solution chosen by the community for this problem is to use *time windows*. When the computation starts, the operator considers only a time-limited subset of the stream, allowing it to give a timely answer to the query/rule.

Moreover, as described in [35], data stream processing engines, in order to perform real-time processing, must ensure compliance with the following eight basic rules:

Keep the data moving. Messages should be processed online, without the need to be stored in any kind of repository before the analysis takes place. The processing model should be active and should not involve any kind of source polling.

Query using stream-enabled SQL. The chosen query language should provide stream-oriented semantic operators and primitives to allow effective querying of the stream.

Handle stream imperfections. The DSMS must be able to handle without issues any kind of imperfection in the stream (e.g., delayed, missing, or out-of-order data) giving the queries consistent answers. Real-world systems have plenty of these issues, making this a crucial feature of the system.

Generate predictable output. The system must ensure that equivalent data sequences in the stream produce the same output results.

Integrate stored and streaming data. A DSMS should allow the seamless integration among streaming and static data (e.g., data stored in a DBMS) possibly allowing to query them adopting the same language. Moreover, the system should

also be able to keep track of its internal state, allowing it to be resumed if necessary.

Guarantee data safety and availability. DSMSs have to ensure high availability since they often operate in critical conditions. Furthermore, they must ensure safe behavior in case of failures.

Automatic partition and scale. The system should be able to transparently distribute its workload among multiple—and different—machines and processors, improving its scalability.

Real-time processing. The engine must be implemented in a highly optimized way introducing a minimal overhead in the processing operation, producing the queries' results in real-time and also in high-data-rate contexts.

The last requirement, together with the consideration that incomplete data can often be tolerated, led to the definition of *load shedding* [5, 37]; this technique is used when the input rate from the data sources exceeds the system capacity, thus causing a system overload and an increase of output latency. In order to fulfill quality-of-service (QoS) constraints and real-time deadlines, some data items (tuples) are discarded. In [37], several techniques are presented for inserting and removing tuple *drop* operators from query plans, either in a random way or following the semantics of the application. The authors discuss *when* and *where* load should be shed and *how much* load to shed from the query plan while maintaining response quality degradation within acceptable limits.

5.3 Data Stream Structure and Constraints on Queries

A stream can assume several different structures, the original definition of stream "an append-only, ordered sequence of timestamped items" [21], having been relaxed in many ways. For instance, if the concept of *revision tuple* is introduced in the language, some tuples are not appended to the stream, but they simply replace other tuples [29].

If the stream is not continuous, but is burst based, it could be viewed as a sequence of sets [38] or, in another possible representation, as a sequence of relational tuples [3] or objects [9, 36].

According to the presence or absence of the timestamp among the attributes of the tuple and to the kind of preprocessing applied before the tuple reaches the system, four different models of stream systems can be defined [19]:

Unordered cash register. Unordered items come from different domains without any kind of preprocessing.

Ordered cash register. Time-ordered items come from different domains without any kind of preprocessing.

Unordered aggregate. Items coming from the same domains are preprocessed and only one aggregate item per domain is appended to the stream without following any particular order.

Ordered aggregate. Items coming from the same domains are preprocessed, and only one aggregate item per domain is appended to the stream following some time-based order.

From the queries' point of view, the property of *monotonicity* ensures that the result of a query can be updated incrementally as data flow in the system; i.e., if $Q(t)$ is the answer to the query Q at time t, executing the query at two different time instants t_i and t_j, $Q(t_i) \subseteq Q(t_j)$, $\forall t_j > t_i$.

Moreover, for monotonic queries, the following property holds:

$Q(t) = \bigcup_{t_i=1}^{t} (Q(t_i) - Q(t_{i-1})) \cup Q(0)$

The property simply states that it is enough to reevaluate the query considering only newly arrived tuples in the stream, appending the new results to the former ones.

On the other hand, *non-monotonic* query results cease to be valid once new items—involved in the query computation—are appended to the stream. Examples of non-monotonic queries are queries that involve *blocking* operators like the NOT (negation) operator that cannot return its answer until all the data to be considered have flown into the system.

5.4 Data Stream Query Languages and Data Stream Management Systems

The main issue encountered while trying to apply the traditional DBMS model to data stream processing is related to the necessity of building a persistent storage where all the relevant data is kept and whose updates are relatively rare (almost true for traditional relational databases but not for data streams). This approach has a negative trend on performance if there are many rules to be processed (more than a given threshold) or when the data arrival rate is too high.

To overcome these limitations, data stream management systems (DSMSs)—a new class of systems oriented toward processing large streams of data in a timely way—have been developed by the database community in different projects and contexts during the last years, having different semantics associated to the queries. The answer to a query can be seen as an append-only output stream or as an entry in a storage that is continuously modified as new elements flow inside the processing stream. An answer can be either exact or approximate, depending on the system memory that has been (and can be) used to store all the required elements of input streams' history [5, 37].

As stated in [16], a DSMS can be modeled as a set of standing queries Q, one or more input streams, and four possible outputs:

The *Stream* is composed of all the elements of the answer that are produced once and never changed.

The *Store* contains the parts of the answer that may be changed or removed at a certain future time point. The stream and the store together define the current answer to Q.

The *Scratch* represents the working memory of the system, where it is possible to store data useful to compute the answer but that is not part of the answer.
The *Throw* is a sort of recycle bin that is used to throw away unneeded tuples.

The model described above is the most complete to define the behavior of DSMSs, showing how the DSMS's only approach cannot entirely cover the needs for IFP (information flow processing), introducing the need of some other kinds of paradigm for complex pattern detection and notification [16, 26].

In the following, a brief presentation of some of those languages and systems is given.

5.4.1 StreaQuel and TelegraphCQ

TelegraphCQ [11] is a general-purpose continuous query system based on a declarative, SQL-based language called *StreaQuel*.

StreaQuel is a relational, SQL-derived stream query language. StreaQuel manages unbounded flows by means of its native window operator *WindowIs*. Several WindowIs operators may be used in a query, one for each input window, embedded in a *for* loop, which is part of each rule and defines a time variable indicating when the rule has to be processed.

The general assumption behind this mechanism is that it must consider an absolute time model. By adopting an explicit time variable, StreaQuel enables users to define their own policy for moving windows. As a consequence, each window may contain a different number of elements, since its dimension is not bounded a priori.

TelegraphCQ is based on a C++ implementation as an extension of PostgreSQL. The compilation of rules into query plans is made through adaptive modules [4, 27, 32], which dynamically decide how to route data to operators and how to order commutative operators. As explained in [33], thanks to this modularity, TelegraphCQ can be spread on multiple machines with a clustered distribution approach, taking into account processing constraints only.

5.4.2 XML-QL and NiagaraCQ

NiagaraCQ [12] is an IFP system for Internet databases, providing a high-level abstraction to retrieve information from a frequently changing environment like Internet sites. NiagaraCQ is based on the *XML-QL* (extensible markup language-query language) [18].

XML-QL is a declarative, relationally complete query language for extensible markup language (XML); its simple structure allows to easily extend well-known

relational database query optimization techniques extracting data from existing XML documents and creating new ones.

The initial draft of this language is presented in a W3C note.[1] In [18], the authors present an extension of XML that allows to manage *ordered* XML documents. This extension introduces in the query language concepts like *tag variables*, *regular path expressions*, multiple-source integration, and *user-defined* functions that allow advanced XML processing.

While the NiagaraCQ engine is designed to run in a totally centralized way, information sources are actually distributed. To increase scalability, NiagaraCQ uses an efficient caching algorithm, which splits operators into groups, keeping members of the same group in the same query plan, thus increasing performance.

5.4.3 OpenCQ

Like NiagaraCQ, *OpenCQ* [25] is an IFP system developed to deal with Internet-scale update monitoring. In OpenCQ, rules are divided into three parts:

- The definition of the operations to be performed on data (in SQL)
- The activation trigger condition
- The stop condition

OpenCQ supports the same processing and interaction models as NiagaraCQ.

OpenCQ has been implemented on top of the DIOM framework [24]: a client-server communication paradigm is used to collect rules and information and to distribute results, keeping the rule processing completely centralized. Wrappers allowing integration among different and heterogeneous sources are used in order to transform the input flow to the required results output flow. Each wrapper can behave in a *pull* or in a *push* way, depending on the source behavior: in the first scenario, the wrapper periodically retrieves data from the source, while in the second, the wrapper simply waits for the source to send in new data that it needs to prepare for processing.

5.4.4 Tribeca

Tribeca [36] has been developed for network traffic monitoring and fits this particular domain. Its rules take a single information flow as input and produce one or more output flows. Rules are expressed using an imperative language, defining the sequence of operators through which the information must flow.

[1] http://www.w3.org/TR/NOTE-xml-ql/

Three different operators exist in the language: selection (called qualification), projection, and aggregate; it is also possible to split (merge) flows by means of ad hoc multiplexing (demultiplexing) operators. Tribeca supports both count-based and time-based windows, which can be sliding or tumble. Windows are used to specify the part of input flows to be processed when performing aggregates. Since items do not have an explicit timestamp, the processing is performed in arrival order. Since the windows are time based, it is impossible to count the number of information items captured at each iteration. Moreover, a rule may be satisfied by multiple and different sets of elements (multiple selection policy), while an element may be part of the computation in more than one iteration, as there is no explicit consumption policy. Rules are translated into direct acyclic graphs in order to be executed, applying graph optimization techniques during the transformation in order to improve performance.

5.4.5 CQL and Stream

Stream computes a query plan starting from *CQL* rules (CQL, *continuous query language*, is an SQL-Like declarative language developed by the database group of Stanford University in the same project), taking into account predefined performance policies in order to schedule the execution of the operators in the plan.

The *transforming rule* is the basic concept of *CQL* [3]; it provides a unified syntax for processing both information flows and stored relations.

CQL provides a clear semantic for the rules, together with a simple language to define them. It defines operators that manage the data stream: due to the fact that a stream is a potentially infinite bag of timestamped data items, these operators extract finite bags of data. The CQL operators split into three different classes as shown in Fig. 5.2:

Relation-to-relation operators derived directly from the SQL operators are the basic, core operators of the language and allow the definition of rules using an SQL-like standard notation (e.g., relational algebraic expressions).

Stream-to-relation operators transform streams in relations; the most well-known operator of this class is the *window* operator that supports flow processing by defining *sliding*, *pane*, and *tumble* windows on the stream.

Relation-to-stream operators (*IStream*, *DStream*, and *RStream*) define how to generate new information flow starting from tuples.

CQL uses a multiple selection policy, associated with a zero consumption policy (the tuples in the stream are never consumed; they still exist after their processing is completed).

Stream provides two shedding techniques to deal with the resource overload problem that is typical of data streams:

- The first one applies load shedding on a set of sliding window aggregation queries to reduce the impact on limited computational resources [5].

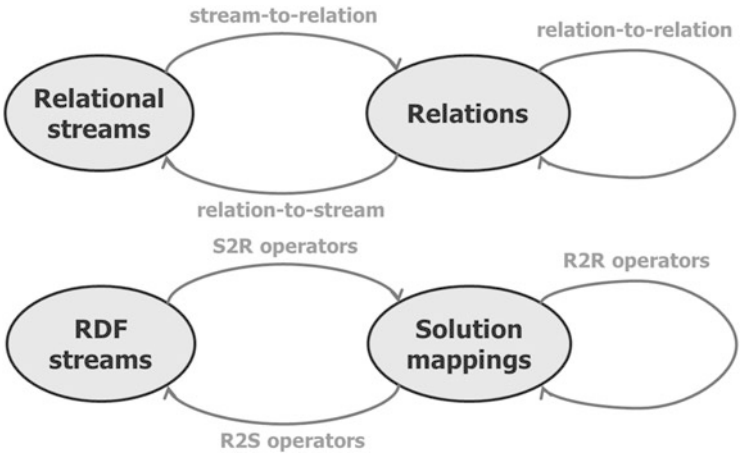

Fig. 5.2 The CQL model

- The second one discards the state of the operator for a set of windowed joins, reducing the impact on memory occupation (in case of limited memory) [34].

The CQL model inspired the design of different Resource Description Framework (RDF) stream processing engines [7, 10, 23], and currently there are different implementations (e.g., C-SPARQL, SPARQLstream, CQELS) of the adapted model (see Chap. 7 [17]). The stream and the relation concepts are mapped to RDF streams and to a set of mappings (using the SPARQL algebra terminology[2]), respectively. To highlight the similarity of RSP operators to those of CQL, similar names are used: S2R, R2R, and R2S to indicate the operators analogous to stream-to-relation, relation-to-relation, and relation-to-stream operators. Chapter 7 [17] presents a description of (1) the sliding window as an example of S2R operators, (2) SPARQL algebra as an example of R2R operators, and (3) the streaming operator as an instance of R2S operators.

5.4.5.1 SQuAl and Aurora/Borealis

Aurora [1] is a general-purpose DSMS that defines transforming rules using an imperative language called *SQuAl*.

SQuAl defines rules in a graphical way, by means of boxes and arrows, connecting different operators of the rule.

SQuAl operates on windows, processing either a single tuple at a time or several tuples at a time, applying a specific user-defined function. This allows information tuples to be used more than once (multiple selection policy), while there exists no consumption policy.

[2]Cf. http://www.w3.org/TR/sparql11-query/

SQuAl allows multiple input processing and multiple result outputs, allowing to restrict the output by means of ad hoc QoS policies in order to fit users and/or application requirements. SQuAl defines the necessary load-shedding policies and customizes the system behavior according to QoS constraints.

Aurora enforces QoS constraints to automatically define shedding policies. Input and output flows do not have an associated timestamp but are processed in their arrival order.

One of the most interesting features of Aurora is the possibility to include intermediate storage points in the rule plan as needed to keep historical information and to recover after operators' failure.

The processing is organized by a scheduler, taking an optimized version of the user-defined plan and choosing how to allocate computational resources to different operators according to their load and to the specified QoS constraints.

Some project extensions have been created in order to investigate distributed processing inside both a single administrative domain and over multiple domains [13]. The goal of these extensions, called *Aurora** and *Medusa*, is that of efficiently distributing the load between available resources; in these implementations, processors communicate using an overlay network, with dynamic bindings between operators and flows.

Recently, the two projects have been merged into the Borealis stream processor [2], which includes all the two previous project features.

5.4.6 GSQL and Gigascope

Gigascope [14, 15] is a DSMS specifically designed for network applications, including traffic analysis, intrusion detection, performance monitoring, etc. Its main concern is to provide high performance for the specific application field it has been designed for. Gigascope is based on a declarative, SQL-like language, called GSQL.

Gigascope translates GSQL rules into basic operators, composing them into a processing plan and using optimization techniques to rearrange the plan according to the nature and the cost of each operator.

GSQL is a declarative, SQL-like language, which includes only filters, joins, group by, and aggregates.

The processing paradigm used by GSQL is very different from other data stream engines. To manage blocking operators, instead of introducing the usual concept of window, it assumes that every input tuple has at least one ordered (monotonically increasing or decreasing) attribute (e.g., the timestamp), using this attribute as a constraint in the processing. For example, the join operator, by definition, must have a constraint on an ordered attribute for each involved stream.

This paradigm has a limited set of application domains (the ones in which it is possible to make strong assumptions on the nature of data and on their arrival order), but it makes the processing semantics easier to understand and similar to that of traditional SQL.

5.4.7 PerLa

The PerLa middleware is a collection of software units designed to support the execution of PerLa queries. Its design revolves around the Functionality Proxy Component (FPC), a proxy object which acts as a decoupling element between sensing nodes and middleware users. In Chap. 4 [30], PerLa is introduced as a pervasive data management language; in this chapter, we briefly introduce the structure of its middleware which, reduced to its essence, constitutes the software environment needed to manage the life cycle of a set of FPCs.

PerLa's middleware provides an easy, XML-based mechanism which allows for a fast integration of new sensors in the system; the designer must only provide a sensor description by means of a standard XML syntax. When the system detects the new sensor descriptor, the parser automatically produces the software implementation of the sensor, and the sensor will be instantly integrated, dispatching its own sensed data as required by the queries.

Being a database-like language, in PerLa the queries' results are automatically stored in a relational database to provide a history of data. In its current development stage, PerLa bases itself on an existing DBMS.

Figure 5.3 shows a high-level snapshot of the whole PerLa system architecture. There are two main runtime layers in the system: a *low-level support* layer, a *hardware abstraction layer*, and the central body of the PerLa middleware itself, which provides standard mechanisms and APIs to handle all the different devices that compose the pervasive system; and a *high-level support* layer that provides query execution services and data management.

Figure 5.4 shows in more detail what the low tier of the middleware is composed and how it operates in order to generate FPCs, while Fig. 5.5 shows the internal structure of a single FPC. Further details about the PerLa middleware are available in [28].

PerLa uses the Channel Manager to create a channel virtualization that allows establishing a bidirectional point-to-point channel between the FPC and the device; this channel can simulate plug and play connections, recognizing new devices when inserted automatically. FPCs are created by the FPC factory, which receives the XML file containing the details about the device and creates an FPC adapter for the node. The FPC registry, on the other hand, maintains the list of all the known devices and their relative FPC.

The PerLa's user interface is constituted of the Query Parser that receives the queries directly from the user (to whom the execution results are provided); after having verified their syntax and validated their semantics, it sends them to the Query Analyzer. The Query Analyzer is responsible for the creation of the low-level query (LLQ) executor that manages the communication with the physical layer and the high-level query (HLQ) executor that is implemented as a wrapper for external data management systems and permits the user to perform data management operations. A LLQ executor is created for each FPC involved, configured for the execution of the statement submitted. FPCs are then consulted for getting data from the physical devices, and results are pushed all together to an output stream.

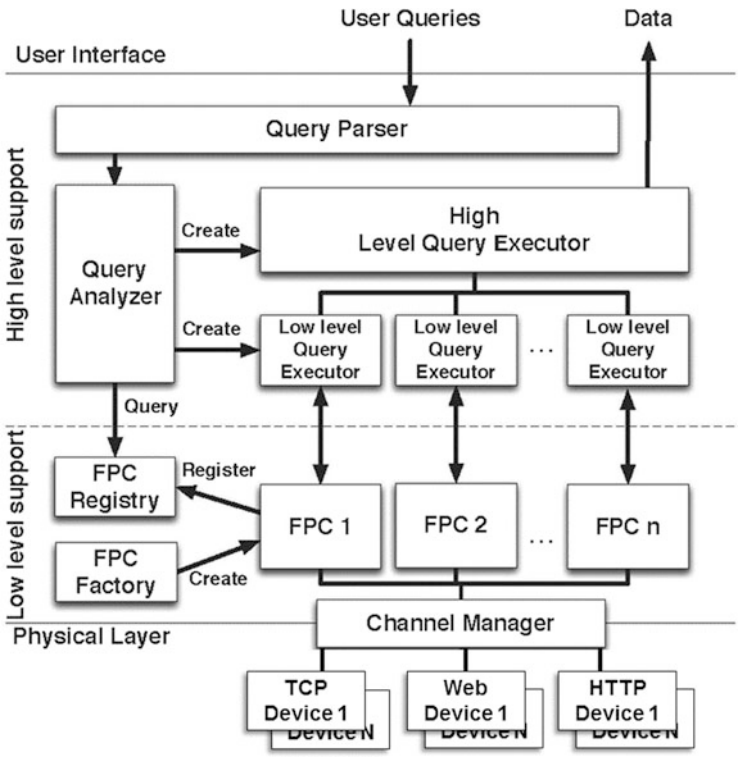

Fig. 5.3 Schema of PerLa's architecture

When nodes are connected to a shared broadcast bus—as in real applications—the channel must be multiplexed: an Adapter Server is added to the middleware to perform (de)multiplexing operations on the FPC side, while on the other side of the connection, an Adapter Client manages the communication channel [8].

The *PerLa query language* [31] is an extension of standard SQL, which includes clauses that allow defining the scope of each of the three kinds of queries:

- HLQ, *high-level queries* which aggregate information coming from different sensors
- LLQ, *low-level queries* which run directly on the sensors to set sampling frequencies and to state how to add data to the stream, possibly performing some kind of aggregation or thresholding on the sensor itself
- AQ, *actuation queries* whose content is a set of commands that activate devices that have capabilities of performing some kind of action (e.g., a small electric engine or a valve)

Further details about the PerLa query language have been presented in Chap. 4 [30].

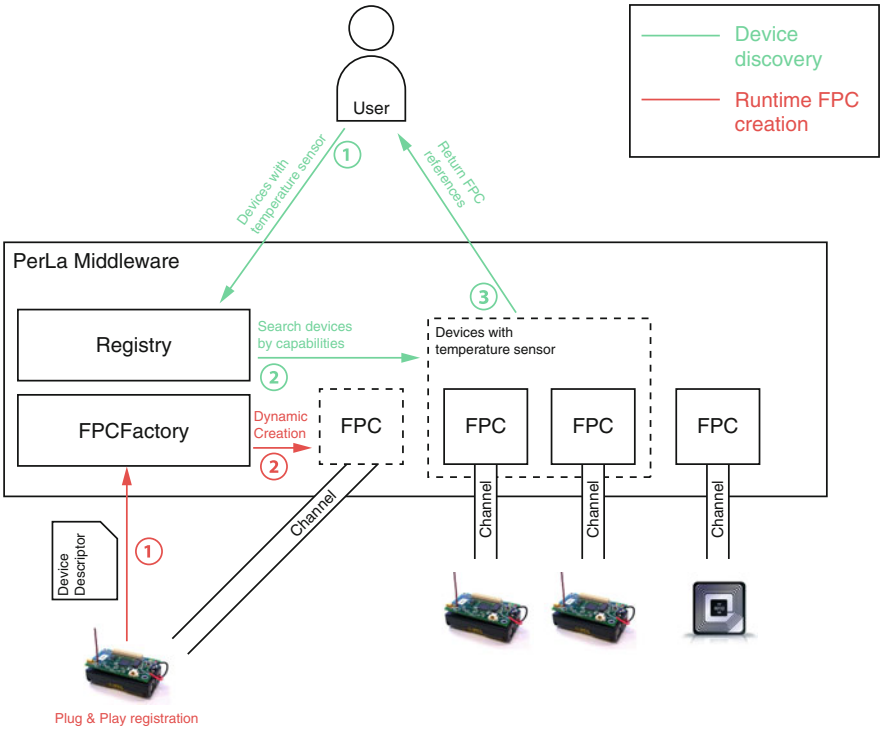

Fig. 5.4 PerLa middleware architecture

5.5 The Stream Mill System

Stream Mill [6] is a general-purpose DSMS that supports an SQL-like continuous query language called Expressive Stream Language (ESL). ESL provides new constructs whereby users can define complex tasks such as data mining tasks and the flow of information between query operators. Therefore, ESL is a highly expressive language which extends the declarative query constructs of SQL with the ability of creating new "user-defined aggregates" (UDAs). UDAs are defined by declaring internal tables and a triplet of statement blocks preceded by the keyword INITIALIZE, ITERATE, and TERMINATE.

The blocks contain update statements that operate on the internal tables. Thus, the statements in INITIALIZE define the operations performed at the beginning of the stream (i.e., when the first tuple is processed), whereas the ITERATE statements define the processing associated with the remaining tuples and TERMINATE specifies the closing computation to be performed after the last tuple. Thus, to define the traditional average aggregate, the user will declare an internal table where the pair $(V, 1)$ is stored by statements in the INITIALIZE block as soon as the first value V of the stream is detected; then, in the ITERATE block, the current pair of values

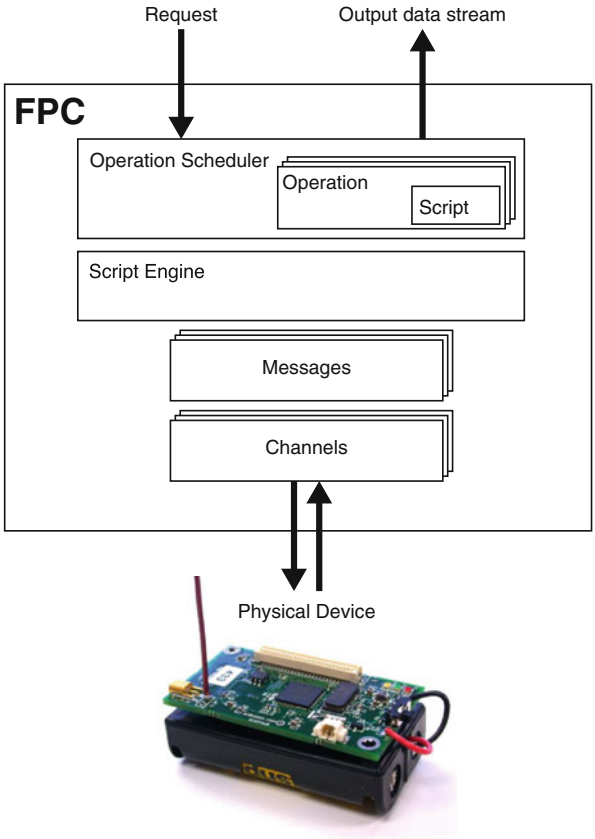

Fig. 5.5 FPC internal architecture

is increased by $(+V, +1)$ for each new arriving input value V. Finally, when the EOF is detected, the statements in the TERMINATE block will compute the ratio between the sum and the count accumulated in the internal table and return it as the final result of the aggregate.

UDAs turn SQL into a Turing-complete language [22] that can express data mining algorithms concisely [39]. However, the UDA just described and every UDA that uses TERMINATE to return its final results are blocking and thus cannot be used on data streams.

A first solution to this problem consists of returning all results in ITERATE, as in the case of cumulative aggregates that produce a new result for each new incoming tuple. As a second solution, SQL-2 aggregates and blocking UDAs can be called on windows whereby, for each new input tuple, they return their value on the whole window—a value that for most aggregates can be computed efficiently from the tuples that just expired out of the window. Thus, ESL allows an additional group of UDA statements to be declared under the EXPIRE label to manage this differential computation.

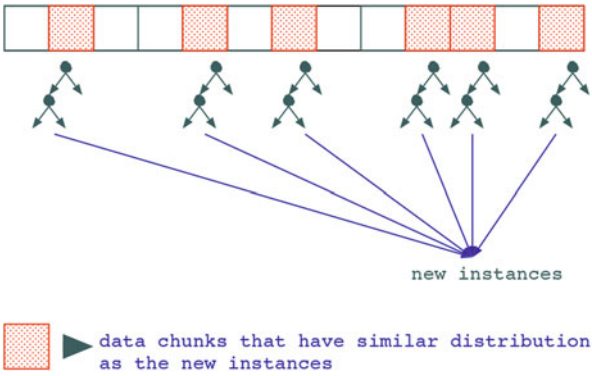

Fig. 5.6 Mining streams with concept drift

These constructs turn ESL into a powerful language for managing a wide range of continuous queries on data streams efficiently, including the search for complex patterns, sketch aggregates, exponential histograms, and approximate frequent item sets [40]. Load shedding techniques that speed up processing by obtaining approximate results can also be expressed through UDAs, which entail the application of popular data mining algorithms to data streams [40].

In fact, using constructs such as window panes, it is often possible to mine a stream considering each chunk of the stream by itself, deriving a model from it and then combining the models as shown in Fig. 5.6 [6]. This approach can also be used to reduce the classification error in environments prone to concept drift, which occurs when the statistical properties of the data changes over time in unpredictable ways lowering the accuracy of the predicted model.

Stream Mill keeps the traditional architecture of a database system; it compiles rules into a query plan whose operators are then dynamically scheduled. The implementation of Stream Mill as described in [6] shows a centralized processing, where the engine exchanges data with applications using a client-server communication paradigm.

Figure 5.7 shows the overall system architecture.

In particular, the system is based on a multiple-client, single-server architecture, allowing the user to define her/his own streams, queries, and aggregates using the client query editor which also displays the processing result to the user. On the server side, the system parses and compiles the continuous queries, generating a query graph that describes which of the tuples have to be extracted from the input buffer and then how they are to be processed in order to be sent to the output buffer. Any UDA is eventually translated in *C++* language, in order to be as fast and efficient as native ESL language operators.

Buffer management policies are adopted to avoid useless waste of main memory, removing tuples from the input buffers as they are processed (except from tuples involved in window-based computation).

Stream Mill also allows the use of different execution policies depending on the user needs: sometimes it is necessary to receive results in real time, so the system

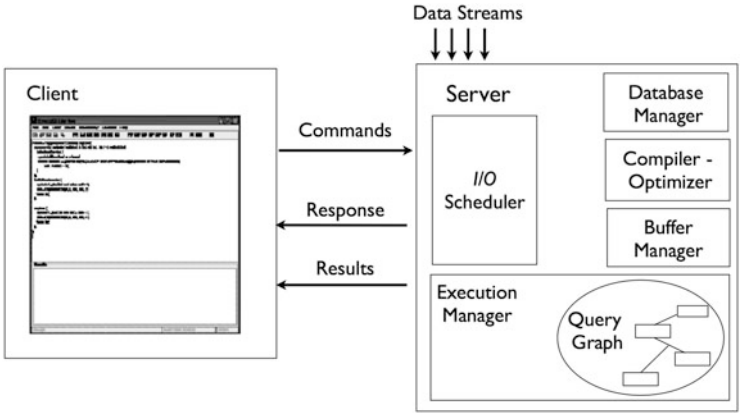

Fig. 5.7 The Stream Mill system architecture

adopts a policy to reduce the response time, while in low-memory scenarios a policy to reduce the main memory occupation will fit best, reducing the necessary memory space to store partial computation results. The switching between different policies is performed at runtime, in a quick and efficient way, without stopping the system query processing.

5.6 Summary and Conclusions

In this chapter, we have introduced some basic concepts on data streams and data stream processing and have reviewed some of the available DSMSs.

In the last section of this chapter, we have also presented in detail *Stream Mill*, a DSMS based on an SQL-like query language providing interesting stream processing features that allow the mining of streams.

The concept presented in this chapter will be the ground on which many of the systems and results presented in the following chapters are built.

References

1. Abadi, D.J., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Erwin, C., Galvez, E., Hatoun, M., Maskey, A., Rasin, A., et al.: Aurora: a data stream management system. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, p. 666. ACM, New York (2003)
2. Abadi, D.J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., et al.: The design of the borealis stream processing engine. In: CIDR, vol. 5, pp. 277–289 (2005)

3. Arasu, A., Babu, S., Widom, J.: The cql continuous query language: semantic foundations and query execution. J. Int. J. Very Large Data Bases **15**(2), 121–142 (2006)
4. Avnur, R., Hellerstein, J.M.: Eddies: continuously adaptive query processing. In: ACM SIGMoD Record, vol. 29, pp. 261–272. ACM, New York (2000)
5. Babcock, B., Datar, M., Motwani, R.: Load shedding for aggregation queries over data streams. In: Proceedings of 20th International Conference on Data Engineering, 2004, pp. 350–361. IEEE, Boston (2004)
6. Bai, Y., Thakkar, H., Wang, H., Luo, C., Zaniolo, C.: A data stream language and system designed for power and extensibility. In: Proceedings of the 15th ACM International Conference on Information and Knowledge Management, pp. 337–346. ACM, New York (2006)
7. Barbieri, D.F., Braga, D., Ceri, S., Valle, E.D., Grossniklaus, M.: Querying rdf streams with c-sparql. SIGMOD Rec. **39**(1), 20–26 (2010). doi:10.1145/1860702.1860705. http://doi.acm.org/10.1145/1860702.1860705
8. Bolchini, C., Orsi, G., Quintarelli, E., Schreiber, F.A., Tanca, L.: Progettazione dei dati con l'uso del contesto. Mondo Digitale (2008)
9. Bonnet, P., Gehrke, J., Seshadri, P.: Towards sensor database systems. In: Mobile Data Management, pp. 3–14. Springer, Heidelberg (2001)
10. Calbimonte, J.P., Corcho, O., Gray, A.J.G.: Enabling ontology-based access to streaming data sources. In: Proceedings of the 9th International Semantic Web Conference on The Semantic Web - Volume Part I, ISWC'10, pp. 96–111. Springer, Heidelberg (2010). http://dl.acm.org/citation.cfm?id=1940281.1940289
11. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S.R., Reiss, F., Shah, M.A.: Telegraphcq: continuous dataflow processing. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 668–668. ACM, New York (2003)
12. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: Niagaracq: a scalable continuous query system for internet databases. In: ACM SIGMOD Record, vol. 29, pp. 379–390. ACM, New York (2000)
13. Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Xing, Y., Zdonik, S.B.: Scalable distributed stream processing. In: CIDR, vol. 3, pp. 257–268 (2003)
14. Cranor, C., Gao, Y., Johnson, T., Shkapenyuk, V., Spatscheck, O.: Gigascope: high performance network monitoring with an sql interface. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, pp. 623–623. ACM, New York (2002)
15. Cranor, C., Johnson, T., Spataschek, O., Shkapenyuk, V.: Gigascope: a stream database for network applications. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 647–651. ACM, New York (2003)
16. Cugola, G., Margara, A.: Processing flows of information: from data stream to complex event processing. ACM Comput. Surv. **44**(3), 15 (2012)
17. Dell'Aglio, D., Balduini, M., Della Valle, E.: Applying semantic interoperability principles to data stream management. In: Data Management in Pervasive Systems, Chapter 7. Springer, Berlin (2015)
18. Deutsch, A., Fernandez, M., Florescu, D., Levy, A., Suciu, D.: A query language for xml. Comput. Netw. **31**(11), 1155–1169 (1999)
19. Gilbert, A.C., Kotidis, Y., Muthukrishnan, S., Strauss, M.: Surfing wavelets on streams: one-pass summaries for approximate aggregate queries. In: VLDB, vol. 1, pp. 79–88 (2001)
20. Golab, L., Özsu, M.T.: Issues in data stream management. ACM Sigmod Rec. **32**(2), 5–14 (2003)
21. Guha, S., McGregor, A.: Approximate quantiles and the order of the stream. In: Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pp. 273–279. ACM, New York (2006)
22. Law, Y., Wang, H., Zaniolo, C.: Relational languages and data models for continuous queries on sequences and data streams. ACM Trans. Database Syst. **36**(2), 8 (2011). doi:10.1145/1966385.1966386. http://doi.acm.org/10.1145/1966385.1966386
23. Le-Phuoc, D., Dao-Tran, M., Parreira, J.X., Hauswirth, M.: A native and adaptive approach for unified processing of linked streams and linked data. In: Proceedings of the 10th International

Conference on the Semantic Web - Volume Part I, ISWC'11, pp. 370–388. Springer, Berlin (2011). http://dl.acm.org/citation.cfm?id=2063016.2063041
24. Liu, L., Pu, C.: A dynamic query scheduling framework for distributed and evolving information systems. In: Proceedings of the 17th International Conference on Distributed Computing Systems, 1997, pp. 474–481. IEEE, Baltimore (1997)
25. Liu, L., Pu, C., Tang, W.: Continual queries for internet scale event-driven information delivery. IEEE Trans. Knowl. Data Eng. **11**(4), 610–628 (1999)
26. Özsu, M.T., Valduriez, P.: Principles of Distributed Database Systems, 3rd edn. Springer, Berlin (2011)
27. Raman, V., Raman, B., Hellerstein, J.M.: Online dynamic reordering for interactive data processing. In: VLDB, vol. 99, pp. 709–720 (1999)
28. Rota, G.: Design and development of an asynchronous data access middleware for Pervasive Networks: the case of PerLa. Master's thesis, Politecnico di Milano (2014)
29. Ryvkina, E., Maskey, A.S., Cherniack, M., Zdonik, S.: Revision processing in a stream processing engine: a high-level design. In: Proceedings of the 22nd International Conference on Data Engineering, 2006. ICDE'06, pp. 141–141. IEEE, Washington (2006)
30. Schreiber, F.A., Roveri, M.: Sensors and wireless sensor networks as data sources: models and languages. In: Data Management in Pervasive Systems, Chapter 4. Springer, Berlin (2015)
31. Schreiber, F.A., Camplani, R., Fortunato, M., Marelli, M., Rota, G.: Perla: a language and middleware architecture for data management and integration in pervasive information systems. IEEE Trans. Softw. Eng. **38**(2), 478–496 (2012). doi:10.1109/TSE.2011.25
32. Shah, M.A., Hellerstein, J.M., Chandrasekaran, S., Franklin, M.J.: Flux: an adaptive partitioning operator for continuous query systems. In: Proceedings of 19th International Conference on Data Engineering, 2003, pp. 25–36. IEEE, Los Alamitos (2003)
33. Shah, M.A., Hellerstein, J.M., Brewer, E.: Highly available, fault-tolerant, parallel dataflows. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, pp. 827–838. ACM, New York (2004)
34. Srivastava, U., Widom, J.: Memory-limited execution of windowed stream joins. In: Proceedings of the 30th International Conference on Very Large Data Bases, vol. 30, pp. 324–335. VLDB Endowment, Toronto (2004)
35. Stonebraker, M., Çetintemel, U., Zdonik, S.: The 8 requirements of real-time stream processing. ACM SIGMOD Rec. **34**(4), 42–47 (2005)
36. Sullivan, M., Heybey, A.: Tribeca: a system for managing large databases of network traffic. In: Proceedings of USENIX (1998)
37. Tatbul, N., Çetintemel, U., Zdonik, S., Cherniack, M., Stonebraker, M.: Load shedding in a data stream manager. In: Proceedings of the 29th International Conference on Very large Data Bases, vol. 29, pp. 309–320. VLDB Endowment, Berlin (2003)
38. Tucker, P.A., Maier, D., Sheard, T., Fegaras, L.: Exploiting punctuation semantics in continuous data streams. IEEE Trans. Knowl. Data Eng. **15**(3), 555–568 (2003)
39. Wang, H., Zaniolo, C.: Atlas: A native extension of SQL for data mining. In: D. Barbará, C. Kamath (eds.) Proceedings of the 3rd SIAM International Conference on Data Mining, San Francisco, 1–3 May 2003, pp. 130–141. SIAM, San Francisco (2003). doi:10.1137/1.9781611972733.12. http://dx.doi.org/10.1137/1.9781611972733.12
40. Zaniolo, C.: Mining databases and data streams with query languages and rules. In: F. Bonchi, J. Boulicaut (eds.) Knowledge Discovery in Inductive Databases, 4th International Workshop, KDID 2005, Porto, 3 October 2005. Revised Selected and Invited Papers. Lecture Notes in Computer Science, vol. 3933, pp. 24–37. Springer, Berlin (2005). doi:10.1007/11733492_2. http://dx.doi.org/10.1007/11733492_2