Extending Relational Query Languages for Data Streams

N. Laptev, B. Mozafari, H. Mousavi, H. Thakkar, H. Wang, K. Zeng, and Carlo Zaniolo

The design of continuous query languages for data streams and the extent to which these should rely on database query languages represent pivotal issues for data stream management systems (DSMSs). The Expressive Stream Language (ESL) of our Stream Mill system is designed to maximize the spectrum of applications a DSMS can support efficiently, while retaining compatibility with the SQL:2003 standards. This approach offers significant advantages, particularly for the many applications that span both data streams and databases. Therefore, ESL supports minimal extensions required to overcome SQL's expressive power limitations—a critical enhancement since said limitations are quite severe on database applications and are further exacerbated on data stream applications, where, e.g., only nonblocking query

N. Laptev · B. Mozafari · H. Mousavi · K. Zeng · C. Zaniolo (✉)
Computer Science Department, University of California, Los Angeles, CA 90095, USA
e-mail: zaniolo@cs.ucla.edu

N. Laptev
e-mail: nlaptev@cs.ucla.edu

B. Mozafari
e-mail: barzan@cs.ucla.edu

H. Mousavi
e-mail: hmousavi@cs.ucla.edu

K. Zeng
e-mail: kzeng@cs.ucla.edu

H. Thakkar
Google Inc., Mountain View, CA 94043, USA
e-mail: hmt007@gmail.com

H. Wang
Sigma Center, Microsoft Research Asia, Beijing 100190, P.R. China
e-mail: haixun.wang@microsoft.com

M. Garofalakis et al. (eds.), *Data Stream Management*,
Data-Centric Systems and Applications, DOI 10.1007/978-3-540-28608-0_18

operators can be used. Thus, ESL builds on user-defined aggregates and flexible window mechanisms to turn SQL into a powerful and computationally-complete query language, which is capable of supporting applications, such as data stream mining and sequence queries that are beyond the application scope of other DSMSs.

1 Introduction

A key research issue for Data Stream Management Systems (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 [1], programming language extensions [2], and an assortment of other solutions provided in publish/subscribe systems [3]. However, the approach of choice for many research projects is to extend SQL and use this tested database workhorse for continuous queries on data streams. Indeed, an SQL-based approach can offer significant benefits, particularly for the many applications that span both data streams and databases. This is because application developers can then use the same language on both streaming data and stored data—rather than having to learn a new language and cope with the resulting impedance mismatch. From this vantage point, it is also clear that data stream constructs should adhere closely to the syntax and semantics of current standards (namely SQL:2003 [4]); indeed the introduction of constructs that are superficially similar to those of SQL, but actually have different syntax or semantics might confuse, rather than help, users writing spanning applications. Therefore, we advocate a conservative and minimalist's approach, which limits SQL extensions to those demanded by the very nature of infinite data streams, and the online, push-based computation model of continuous queries. Indeed Data Stream Management Systems (DSMSs) must operate as follows:

(a) Results must be pushed to the output promptly and eagerly, while input tuples continue to arrive—i.e., without waiting for (i) the results to be requested by other applications (e.g., a procedural program embedding the continuous query), and (ii) the end of input streams (when blocking operators can be finally applied).
(b) Because of the unbounded and massive nature of the data streams, all past tuples cannot be memorized for future uses. Only synopses, such as windows, can be kept in memory and the rest must be discarded.

The main problem created by (a) is a significant loss of expressive power. Database researchers have long been aware of expressive power limitations of SQL and other relational languages; in fact, this problem has motivated much research on topics such as recursive queries, database mining queries, sequence queries, and time-series queries. Unfortunately, these limitations are dramatically more marring on 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 [5, 6],

2. Database extenders (i.e., libraries of external functions written in procedural languages using BLOBs and CLOBs to exchange data) that have successfully enhanced the versatility of Object Relational (OR) DBMS are much less effective in DSMS, which process data at small increments rather than as aggregate large objects,
3. Embedding queries in a procedural programming language, a solution used extensively in relational database applications, is now of very limited effectiveness on data streams.

We will now expand on these statements starting from point 1. A blocking query operator is one that only returns answers when it detects the end of its input, while a nonblocking operator produces its answers incrementally as new input tuples arrive [5]. For continuous queries, the users must see the results immediately and incrementally as new stream records arrive, rather than when the stream eventually ends: thus only nonblocking query operators are allowed on data streams [5, 6].

The nature of nonblocking queries was formally characterized in [6], where it was shown that only monotonic queries[1] can be expressed by nonblocking operators [6]. Database query languages contain many nonmonotonic (and therefore blocking) query operators/constructs: for instance, set difference and division are nonmonotonic operators in relational algebra, while constructs such as EXCEPT, NOT EXIST, and traditional aggregates in SQL are nonmonotonic. Then, a natural question is whether all the monotonic queries expressible in a query language can be expressed using only its monotonic operators or constructs. A query language that satisfies this criterion is said to be *NB-complete*—i.e., complete for nonblocking queries [6]. A query language that is NB-complete is as suitable a query language for data streams as it is for databases (since by disallowing its nonmonotonic operators/constructs we only lose queries that are of not suitable for continuous online answering). Unfortunately, both relational algebra and SQL-2 fail the NB-completeness test [6]. Thus, the banishment of blocking operators and lack of NB-completeness further aggravate the expressive power limitations of SQL that has made it difficult for DBMS to support new application domains. This problem is discuss next.

As outlined in point 2 above, following the introduction of a time-series datablades by Illustra [7], OR DBMSs offered a plethora of such libraries covering a wide spectrum of applications under an assortment of different names. These functions often use large objects (BLOBs and CLOBs) to exchange data with SQL: for instance, a whole sequence could be encoded as a BLOB and shared between the database and the datablade. This solution is less suitable for data streams, where the computation must proceed continuously in small increments—e.g., by processing each new tuple in the sequence, rather than having to wait for it to be assembled into a BLOB. Indeed, unlike in OR DBMS, datablades have not played an important role as an extension mechanism for DSMS.

[1] Queries are viewed as mappings from the database, or the data stream, to the query answer.

Point 3 above considers the solution of embedding SQL queries in a procedural programming language (PL); currently this represents a commonly used approach for developing complex database applications, since the application logic that cannot be expressed in SQL can then be easily implemented in the PL. This solution, however, uses cursors and get-next constructs; thus it relies on a pull-based computation model that loses much of its effectiveness in the push-based environment of data streams. Indeed in the continuous environment of data streams, consumption (production) of input (output) tuples follows a push-based mechanism—i.e., production (consumption) occurs without waiting for get-next requests from an embedding procedural language.

The severe limitations resulting from the problems discussed in points 1–3 above suggest that, for any SQL-based DSMS, extensions to enhance the power and flexibility of its query language are crucial to compete against the many alternatives proposed, including [1–3, 8, 9] that are discussed in Sect. 7.

Fortunately, a solution to SQL's expressivity and flexibility problems is at hand: user-defined aggregate functions adopt a computation model based on incremental additions to their input streams, rather than the large BLOB objects of datablades. As discussed in [10, 11], User-Defined Aggregates (UDAs) can be written in an external programming language or natively in SQL itself: the native UDA definition capability makes SQL Turing-complete and NB-complete, i.e., able to express every monotonic function expressible by a Turing Machine [6]. We can finally return to the problems caused by the unbounded nature of data streams, mentioned in (b) above, and observe that windows have proved by far to be the most popular form of synopses used in DSMSs. Windows are actually not new to database languages, since SQL-2003 supports logical and physical windows on a set of built-in aggregates called OLAP functions. Many DSMSs also support these windows, and actually introduced new ones, such as slides and tumbles [1, 12], for their built-in aggregates. However, for a DSMS to address both problems (a) and (b) above effectively, its language should support these windows on arbitrary UDAs besides on built-in aggregates. Indeed, the Stream Mill system developed at UCLA supports (i) windows on arbitrary UDAs and (ii) the native definition of these UDAs in SQL [13]. Thus Stream Mill is unique in this respect, and its uniqueness is reflected in the much broader range of applications it can support efficiently: for instance, its Expressive Stream Language (ESL) can express data mining queries, sequence queries, approximate queries, etc., that are beyond the reach of other DSMSs. Examples of these advanced applications will be discussed later in this chapter, which is organized as follows.

In Sect. 2, we introduce the main extensions to SQL:2003 supported in ESL, including UDAs. In Sect. 3, we extend ESL to support different kinds of windows on arbitrary UDAs. Then, in Sects. 4 and 5, respectively, we demonstrates the effectiveness of these extensions in expressing approximate computations and advanced data mining algorithms. In Sect. 6, we describe the architecture of the Stream Mill system that supports these advanced query constructs very efficiently. Section 7 contains a broad description of related work, and it is then followed by the conclusion.

2 ESL: An Expressive Stream Language Based on SQL

ESL supports ad-hoc SQL queries and updates on database tables and continuous queries on data streams. Each data stream is declared by a CREATE STREAM declaration that also specifies the external wrapper from which data is imported and the timestamp associated with the stream. For instance, in Example 1, the data stream **OpenAuction** is declared as having **start_time** as its external timestamp.

Example 1 Declaring Streams in ESL

```
CREATE STREAM OpenAuction(
    itemID INT, sellerID CHAR(10),
    start_price REAL, start_time TIMESTAMP)
  ORDER BY start_time SOURCE ... /* Wrapper ID */;
```

In ESL, new streams can be defined from existing streams in ways similar to defining virtual views in SQL. For instance, to derive a stream consisting of the auctions where the asking price is above 1000, we can write:

Example 2 Performing Selection Operations on Streams

```
CREATE STREAM expensiveItems AS
SELECT itemID, sellerID, start_price, start_time
FROM OpenAuction WHERE start_price > 1000
```

In terms of semantics, these ESL operators produce the same results as if instead of being applied to data streams, they were applied to database tables to which new tuples are being continuously appended. Additional operators supported by ESL on data streams are (i) aggregates (built-in or user-defined) (ii) joins of a stream with a database table, and (iii) union of two or more data streams. All the operators considered so far operate on a single data stream, except for union which always returns tuples sorted by timestamp (thus its equivalent SQL statement is really UNION ALL followed by ORDER BY TIMESTAMP).

In [6], it was proven that constructs mentioned above make ESL NB-complete, thus capable of expressing all nonblocking computations on data streams. In the rest of the chapter, we therefore concentrate on these constructs, which are the most distinctive feature of ESL, and their effective use in expressing complex data stream applications. Space limitations prevent us from covering here other constructs, such as window joins, inasmuch as these constructs are less critical in terms of expressive power, and their treatment by ESL is similar to that of other DSMSs [14, 15].

2.1 User-Defined Aggregates (UDAs)

In recent years, some of the most successful SQL extensions involve aggregates, including (i) data cubes, and (ii) OLAP functions where continuous aggregates are

incrementally computed on logical and physical windows [4]. Logical and physical windows for aggregates are also provided by many DSMSs, some of which also support additional window constructs based on the notions of slides and tumbles [1, 12]. Many current DBMSs and DSMSs also allow the importation of new UDAs defined in external programming languages; however they do not support windows on such UDAs. Now, ESL brings two important improvements to the state-of-the-art by supporting (i) windows on arbitrary UDAs—including logical windows, physical windows, tumbles, and slides—and (ii) the definition of UDAs in SQL (besides external PLs).

Example 3 declares a UDA, myavg, equivalent to the standard **avg** aggregate in SQL, using a definition consisting of the three statement groups that are labelled INITIALIZE, ITERATE, and TERMINATE. Commercial DBMSs [9] and DSMSs supporting UDAs [1, 15] allow a programmer to use external procedural languages to specify the computations to be performed in INITIALIZE, ITERATE, and TERMINATE; this capability is also available in ESL, which however also supports native UDA definition.

In Example 3, the first line in the UDA definition declares a local table, **state**, to keep the sum and count of the values processed so far. Then, the INITIALIZE statement inserts the value taken from the input stream and sets the count to 1. The ITERATE statement updates the table by adding the new input value to the sum and 1 to the count. The TERMINATE statement returns the ratio between the sum and the count as the final result of computation using the INSERT INTO RETURN statement. Myavg and similar UDAs can then be used as standard SQL aggregates, with optional GROUP BY clause.

Example 3 Defining the standard aggregate average

```
AGGREGATE myavg(Next Int) : Real
{    TABLE state(tsum Int, cnt Int);
     INITIALIZE : {
        INSERT INTO state VALUES (Next, 1);
     }
     ITERATE : {
        UPDATE state
           SET tsum=tsum+Next, cnt=cnt+1;
     }
     TERMINATE : {
        INSERT INTO RETURN
           SELECT tsum/cnt FROM state;
     }
}
```

Natively defined UDAs provide an extensibility mechanism of great power and flexibility; in fact, SQL with natively defined UDAs becomes Turing complete, and thus can express all computable queries on database tables [6]. Aggregates such as

myavg, however, are blocking and thus do not add to the expressive power of SQL in data stream applications, which instead require nonblocking aggregates.

Basically, there are two ways to turn a UDA such as **myavg** into a nonblocking UDA. The first is to modify its definition so it becomes a continuous aggregate that returns values during, rather than at the end of, the computation.

Example 4 The continuous average: a nonblocking UDA

```
AGGREGATE online_avg(Next Int) : Real
{    TABLE state(tsum Int, cnt Int);
     INITIALIZE : {
        INSERT INTO state VALUES (Next, 1);
     }
     ITERATE: {
        UPDATE state
           SET tsum=tsum+Next, cnt=cnt+1;
        INSERT INTO RETURN
           SELECT tsum/cnt FROM state
           WHERE cnt % 200 = 0;
     }
     TERMINATE : { }
}
```

For instance, Example 4, shows a continuous version of average where results are returned every 200 tuples rather than at the end. Observe that in this definition the TERMINATE state is empty which assures that the aggregate is nonblocking and monotonic.[2]

The second way to deal with a blocking aggregate consists of keeping the original definition unchanged and using it to continuously recompute the aggregate on a sliding window—in a fashion similar to that of SQL:2003 OLAP functions. We next illustrate an interesting application of the first class of nonblocking aggregates; window-based applications will be discussed in the next section.

2.2 Pattern Queries

Since UDAs process tuples one-at-a-time, they are effective on physically-ordered sequences and can search for patterns in a sequence effectively. Say, for instance, that we want to find the situation where users, immediately after placing an order, ask for a rebate and then cancel the order. Finding this pattern in SQL requires two self-joins on the incoming event-stream of user activities. In general, recognizing the

[2] Updates and other nonmonotonic constructs can still be used freely on other database tables, such as state. But any operator applied to the data stream must be monotonic.

pattern of n events would require $n-1$ joins and queries involving the joins of many streams can be complex to express in SQL. Furthermore, such queries would be very inefficient on data streams. In particular, the condition that a tuple must *immediately follow* another tuple is complex and inefficient with basic SQL but easy with UDAs. For instance, consider an incoming stream of purchase actions:

webevents(CustomerID, ItemID, Event, Amount, Time)

We want to detect the pattern of an order, followed by a rebate, and immediately after that, a cancellation of the same item. Then the following nonblocking UDA can be used to return the string 'pattern123' with the CustomerID whose events have just matched the pattern (the aggregate will be called with the group-by clause on **CustomerID, ItemID**). This UDA models a finite state machine, where 0 denotes the failure state that is set whenever the right combination of current-state and input is not observed. Otherwise, the state is first set to 1 and then advanced step-by-step to 3, where 'pattern123' is returned and the computation continues to search for the next pattern.

Example 5 First the order, then the rebate, and finally the cancellation

```
AGGREGATE pattern(CustomerID Char, Next Char) : (Char, Char)
{    TABLE state(sno Int);
       INITIALIZE : {
          INSERT INTO state VALUES(0);
          UPDATE state SET sno = 1 WHERE Next= 'order';}
       ITERATE: {
          UPDATE state SET sno = 0
            WHERE NOT (sno = 1 AND Next = 'rebate')
              AND NOT (sno = 2 AND Next = 'cancel')
              AND Next <> 'order'
          UPDATE state SET sno = 1 WHERE Next= 'order';
          UPDATE state SET sno = sno+1
            WHERE (sno = 1 AND Next = 'rebate')
              OR (sno = 2 AND Next = 'cancel')
          INSERT INTO RETURN
            SELECT CustomerID,'pattern123' FROM state WHERE sno = 3;
    } }
```

While UDAs can be effectively used to search for simple patterns, this approach can easily become prohibitive for more involved patterns.

Fortunately, powerful languages based on Kleene-* constructs were recently to facilitate the expression of complex sequential patterns. The first such language was SQL-TS [16, 17], for which powerful query optimization techniques were also developed [16, 17]. This led to an industrial SQL-change proposal [18] which has been prototyped more recently [19]. Recently proposed query languages based on Kleene-* constructs include SASE [20], SASE+ [21], Cayuga [22], CEDR [23], and

finally, K*SQL [24, 25]. Stream Mill supports K*SQL, which is the most powerful of these languages, inasmuch as it can express complex queries over relational data, and complex XML streams. K*SQL uses nested word automata[3] which in turn are implemented as UDAs. For instance, the following K*SQL query could be implemented by calling UDA in Fig. 5 with a PARTITION BY **CustomerId** (where ORDER BY **Time** is implicitly assumed for our timestamp-ordered data streams).

Example 6 (Example 5 expressed in K*SQL)

```
SELECT 'modified-pattern123', X.CustomerId
FROM webevents
        PARTITION BY CustomerId
        AS PATTERN (X Y Z)
WHERE
        X.Event = 'order' AND
        Y.Event = 'rebate' AND Y.ItemID = X.ItemID AND
        Z.Event = 'cancel' AND Z.ItemID = Y.ItemID
```

Thus we have a simple pattern specifying the sequence of 3 events where the second (third) event immediately follows the first (second) in the separate CustomerId substream. Languages such as SASE+ [21] adopt a semantics where the next event in the pattern is simply required to follow, rather than immediately follow, the previous event. This more relaxed pattern can be expressed in K*SQL by simply changing the pattern in Example 6 to AS PATTERN (**X V* Y W* Z**). Here V* and W* each can match zero or more successive events,[4] thus a clause such as **Z.Time-Y.Time** $\leq$ **60** might be advisable to limit the overall times elapses to 60 minutes. By adding $F.$ in the `WHERE` clause, we can express local conditions, i.e., F.Z.ItemID = F.Y.ItemID will only require that in each occurrence of F the ItemID is the same, while different occurrences of F can have different values of ItemID.

This seemingly simple extension, makes K*SQL strictly more expressive than its counterparts. Fortunately, despite its higher expressive power, K*SQL has also proved to be highly amenable to efficient execution over high-volumes of stored sequences and data streams [25]. For these reasons, and due to the appealing syntax of K*SQL for sequence queries, the Stream Mill system also supports special built-in UDAs that can efficiently execute K*SQL queries [24]. Likewise, it was previously shown that queries expressed in SQL-TS [16] could be mapped into equivalent ESL queries through the use of specialized UDAs [28]. Similarly, UDAs were used in [28] to implement the FSA computation used by Yfilter to support multiple queries on streaming XML data, thus unifying the processing of these two kinds of streams.

Thus, UDAs have proved to be a key extension of SQL. Furthermore, ESL greatly enhances their power and versatility for data stream applications by providing the flexible window mechanisms for arbitrary UDAs that are discussed next.

[3] Nested words [26] and visibly-pushdown automata [27] can model data with both sequential and hierarchical structures, such as XML, RNA sequences or procedural software traces.

[4] However, expressing "immediately following" in SASE+ is significantly more difficult.

3 Window Aggregates and Their Applications

Following SQL:2003, ESL uses the OVER clause to specify (i) the type of window (i.e., logical or physical), (ii) the size of the window (using a time span for logical windows or the number of tuples for physical ones), and (iii) the columns in the partition-by clause (if any). However, for data streams, the ORDER BY clause can be omitted, since data streams are always ordered by their timestamps.

In Example 7, we have a physical window of 100 items, consisting of the current tuple and the 99 rows preceding it; a separate window is maintained for each seller, as specified by the (PARTITION BY **sellerID**) clause.

Example 7 For each seller, maintain the max selling price over the last 100 items sold.

```
CREATE STREAM LastTenAvg
SELECT sellerID, max(price) OVER
        (PARTITION BY sellerID ROWS 99 PRECEDING)
FROM ClosedPrice;
```

By replacing, say, 'ROWS **99**' with 'RANGE **5** MINUTES', we would instead specify a logical window of five minutes.

Because of their many uses, the notions of window slides and tumbles [1, 12], are now supported in many DSMSs, although they go beyond the SQL:2003 standards. These constructs are supported in ESL for arbitrary UDAs, using a 'SLIDE' declaration in the window clause. For instance, the following example is similar to the previous one in every aspect, but the fact that results are now returned every 10 rows, rather than after each row as in the previous case.

```
CREATE STREAM LastTenAvg
SELECT sellerID, max(price) OVER
        (PARTITION BY sellerID ROWS 99 PRECEDING SLIDE 10)
FROM ClosedPrice;
```

In this example, the size of the slide (10) is smaller than the overall size of the window (100). Tumbles instead occur when the size of the slide exceeds that of the window. For instance, to break the input stream into blocks of size 600 and return the average of the last 600 input tuples at the end of each block, we can specify ROWS **599** PRECEDING SLIDE **600**. Thus the size of the window and the slide are both 600, and the input data stream is partitioned into windows of size 600 and results are returned every 600 tuples.

Window Aggregates

Providing efficient, integrated management and support for an assortment of different windows represents an interesting research problem that in the past has

been addressed only for specific built-in aggregates or specific classes of aggregates [1, 12, 29]. A naive approach to implement windows on arbitrary UDAs requires the user to write six versions of an aggregate, one for each combination of (logical|physical × tumble|slide|no-slide). With ESL, however, users only need to write at most the following two versions: (a) the base version of the UDA, and (b) an **optional** window-optimized version. The second version (b) allows the user to perform delta-maintenance on arbitrary UDAs, which can lead to significant performance improvement. The ESL compiler utilizes these two definitions to provide integrated support for (i) general optimization tasks that are applied to all window aggregates, along with (ii) user-specified optimizations that are specified for a particular UDAs. We next illustrate how this is accomplished using the MAX aggregate as an example whose base definition is shown below:

Example 8 Base Definition for MAX

```
CREATE WINDOW AGGREGATE max (Next Real) : Real
{    TABLE current(CVal real);
     INITIALIZE : {
        INSERT INTO current VALUES (Next);
     } /* the value in is the first max */
     ITERATE : {
        UPDATE current set CVal = Next
           WHERE CVal < Next;
        INSERT INTO RETURN
           SELECT CVal FROM state;
     }
}
```

The Stream Mill system provides uniform support for physical and logical windows, thus the six combinations above degenerate to three, which we discuss next. The simplest one is that of tumbles, i.e., the case in which the UDA is called over a window of size smaller or equal to that of its slide. For instance, say that MAX is called on a window of 600 tuples and the size of its slide is S $\geq$ 600. In this case, ESL will use the base definition of MAX in Example 8 and return the result after 600 tuples. Then the next $S - 600$ input tuples are ignored and the computation restarts from $(S - 600 + 1)$th tuple and goes for another 600 tuples, and so on. Examples illustrating the use of tumbles in clustering and ensemble-based classification are given in Sects. 5.1 and 5.2.

The second case is that of Example 7, in which the UDA is called without the slide construct. A naive implementation consists in buffering all the window tuples into an inwindow table.[5] Then, for each arriving tuple, base MAX aggregate is recomputed over the tuples in inwindow table. While this approach is correct, it is

[5]Newly arriving tuples are inserted into and expiring tuples are removed from the inwindow table automatically by the system for efficiency.

obviously inefficient. Thus ESL allows users to define a specialized version of the UDA which uses the values of tuples leaving the window to perform delta maintenance. For an aggregate such as sum, this delta maintenance involves the subtraction of the expiring value from the current sum. For MAX and more sophisticated aggregates the delta maintenance is more complex but nevertheless quite beneficial. As shown in Example 9, the delta maintenance is performed in a special state called EXPIRE. In our window version of **MAX**, shown in Example 9, the EXPIRE event does not require any action. The expiring tuple is simply discarded (automatically, by the system). The ITERATE state of the UDA only keeps tuples that can potentially be the maximum in the window. Thus, the oldest tuple in the inwindow table is the maximum in the current window.

The result of the aggregate is the same whether this delta computation is performed as soon as a tuple expires, later when a new tuple arrives, or anywhere in between these two instants. ESL takes advantage of this freedom to optimize execution.

Example 9 MAX with Windows

```
CREATE WINDOW AGGREGATE max (Next Real) : Real
{    TABLE inwindow(wnext real);
     INITIALIZE : {
        INSERT INTO RETURN VALUES (Next);
     } /* the system adds new tuples to inwindow */
     ITERATE : {
        DELETE FROM inwindow WHERE wnext ≤ Next;
        INSERT INTO RETURN VALUES (oldest());
     }
     EXPIRE: {  } /*expired tuples are removed automatically*/
}
```

In the definition of window aggregates, EXPIRE is treated as an event that occurs once for each expired tuple—and the expired tuple is removed as soon as the EXPIRE statement completes execution. ESL also provides the built-in predicate **oldest()** which selects the oldest tuple among the tuples of **inwindow**: **oldest().wnext** delivers the **wnext** column in this tuple. If the tuple has only one column then the system allows using **oldest()**, i.e., without the column name.

Upon arrival of a new tuple, the system first proceeds at executing any outstanding EXPIRE events. The ITERATE statements are next executed on this newly arrived tuple. After the ITERATE statements, the new tuple is put into the **inwindow** buffer.

This delta-maintenance approach to window aggregates is also used in the implementation of basic ESL built-in aggregates. For instance, in case of the MAX aggregate, we eliminate tuples in the buffer that are dominated by more recent tuples—thus reducing the size of the buffer from W to $\log(W)$, where W denotes the size of the window. Frequently, the approach is also effective on more complex UDAs, such as the approximate frequent-items example discussed in Sect. 4.

For the third case, i.e., where the specified slide size is less than the window size, ESL utilizes the optimization that has been proposed in [12]. This optimization involves dividing the window in smaller panes. Window version of the UDA is used to perform the delta computation on the results of the base version of the UDA on each smaller pane [30]. Using these powerful UDAs we can also implement the 'negative tuple' semantics[6] that has also been considered for windows [31].

The introduction of powerful analytics SQL:2003 have illustrate the need for DBMS to support a wider range of aggregates than those supported in SQL-2, and the important role that windows play in this context. However, windows aggregates play an even more important role in DSMS particularly those that must support complex tasks with QoS guarantees: in fact, window UDAs can support effectively (i) approximate computations, (ii) load shedding, and (iii) complex decision support and mining task. Our discussion in the rest of the chapter will focus on these topics.

4 Approximation and Sketch Aggregates

In order to assure QoS against high arrival rates and bursts in the incoming data streams, DSMS rely on (i) load shedding, and (ii) approximation techniques, which dovetail with and UDA-oriented architecture. Indeed, Stream Mill provides an architecture where load shedding is integrated with UDAs [32], supporting error models that accommodate different requirements for multiple users, different sensitivities to load shedding, and different penalty functions. By incorporating a priori statistics of data streams, the Stream Mill system can provide QoS guarantee for a large class of queries, including traditional SQL aggregates, statistical aggregates and data mining functions [32]. A Bayesian approach can be used to combine the past statical information about query answers and further boost the quality of the a priori estimations [33].

We now leave the discussion of load-shedding which would take us beyond the scope of this paper to concentrate on sketches and other approximate aggregates.

EH Sketches

Many data mining techniques require counting the frequency of different items in a window. Since computing the exact counts would require storing the whole window to determine the tuples leaving the window. For windows that are too large in size approximate counting aggregates can be used instead. In [34], Datar et al. proposed the Exponential Histogram (EH) sketch algorithm for approximating the number of 1's in sliding windows of a 0–1 stream and showed that for a δ-approximation of the

[6]Besides returning values when new tuples arrive, under such semantics, new revised aggregate values could be produced as soon as a tuple expires out from the time-based window.

number of 1's in the current window, the algorithm needs $O(\frac{1}{\delta}\log W)$ space, where W is the window size.

The EH sketch consists of an ordered list of buckets or *boxes*. Every box in an EH sketch basically carries on two types of information, a time interval and the number of observed 1's in that interval. The intervals for different boxes do not overlap and every 1 in the current window should be counted in exactly one of the boxes. Boxes are sorted based on the start time of their intervals. For each new coming 1, EH creates a new box with size one. Then the algorithm checks if the number of boxes with the same size exceeds $k/2+2$ (where $k=\frac{1}{\delta}$), it merges the oldest two such boxes. The merge operation adds up the size of the boxes and merges their intervals. The final estimation of the number of 1's would be the aggregate size of boxes minus half of the oldest box's size. Note that before reporting the results, we discard the boxes which do not overlap the current window. Thus an EH UDA can be expressed by the ESL code below, and then called in a way similar to built-in SQL3 aggregates.

Example 10 Counting by Exponential Histograms

```
STREAM zeroOnes(val Int, t Timestamp);
AGGREGATE EHCount(next Int, t Timestamp, k Int):{
    WINDOW EH(h Int, t Timestamp) ORDER BY t;
    TABLE memo(last Int, total Int) MEMORY VALUES (0,0);
      AGGREGATE merge(next Int, t Timestamp, k Int):{
        /* state table stores the current box, count, and the last two timestamps */
        TABLE state(h Int, cnt Int, t1 Timestamp, t2 Timestamp) MEMORY;
        INITIALIZE:
        {INSERT INTO state VALUES(next, 1, t, NULL);}
        ITERATE:{
          UPDATE state SET cnt=cnt+1, t2=t1, t1=t WHERE h=next;
           /* should early return if true */
          UPDATE state SET h=next, cnt=1, t1=t
              WHERE h <> next AND (SELECT cnt FROM state) < k/2+2;
          /* should early return if true */
        /* if current count is k/2+2, delete the last box, and double the next-to-last one */
          DELETE FROM EH h WHERE h.t = (SELECT t1 FROM state)
                  AND h <> next AND (SELECT cnt FROM state) = k/2 + 2;
          UPDATE EH h SET h=h*2 WHERE SQLCODE = 0
                  AND h.t = (SELECT t2 FROM state);
          UPDATE state SET h=next, cnt=2, t1=t WHERE SQLCODE = 0; }
      }; /* the end of merge aggregate */

    INITIALIZE:ITERATE:{
      INSERT INTO EH VALUES(next, t) WHERE next > 0; /* ignore 0's */
      UPDATE memo SET total = total + next;
      SELECT merge(h, t, k) OVER (ORDER BY t DESC) FROM EH;
      /* Update last pointer due to merge */
      UPDATE memo SET last = (SELECT max(h) FROM EH);
      INSERT INTO Return SELECT total FROM memo;}
```

```
    EXPIRE:{
      UPDATE memo SET last = h/2
        WHERE (SELECT count(1) FROM EH h WHERE h.h = last) =1;
      /* update total pointer */
      UPDATE memo SET total = total-h/2-last/2 }
    }
      /*Calling the aggregate just defined from an ESL statement /*
SELECT EHCount(val, t, 2) OVER (RANGE 10 MINUTE) FROM zeroOnes;
```

This sketch is later used in several other structures and algorithms listed in [35]. Recently, [36] has used EH to generate an approximate B-bucket equi-depth histogram for data streams with sliding windows. The proposed approach which is called BAr-Splitting Histograms (BASH) is based on dividing the acceptable input range into several chunks or *bars* in a way that each bar contains roughly equal number of items. The number of bars are limited to fix value which is greater than B for improving the accuracy. To keep the size of bars roughly the same as the stream passes, a splitting/merging technique is employed to split big bars, and merge adjacent small bars. BASH provides a very fast and space-efficient equi-depth histogram particularly for high speed data streams.

Approximate Frequent Items

The problem of determining the frequent items in a data stream is important in many applications and several algorithms have been proposed to deal with the common situation where there is enough memory for the frequent items but not for all items [37–39]. Here we focus primarily on [37], which is a windowed approximate frequent items algorithm suitable for delta computation. The algorithm, shown with ESL code in Example 11, maintains k hash-tables over the current window. Each hash-table has a corresponding hash-function. Each hash entry in the hash-tables is an integer, which is used as a counter. When an item enters the window, we iterate through the k hash-functions and determine the k key values. For each key value, we increment the counter at that location in the corresponding hash-table. Similarly, when an item expires out of the window, we decrement the corresponding k counters. Finally, the approximate frequency of an item is determined by taking the minimum value of the k counters. Note, that this minimum value may over estimate the frequency of the item, if all k keys have at least one other item mapped to it. This algorithm can also be viewed as a bloom-filter with two exceptions: (i) there are k different hash-tables instead of just one and (ii) each entry is an integer(counter) as opposed to a bit. In addition to the delta maintenance property, the algorithm provides bounded error estimates. Thus, given the available amount of memory we can estimate the expected error.

Example 11 Approximate Frequency Count

```
STREAM items(item Int); /* Stream of items */
TABLE hash_tables(index1 Int, index2 Int, cnt Int) MEMORY;
  /* the k hash tables index2 goes from 1 to k*/
TABLE hs(h Int, ah Int, bh Int) MEMORY; /* constants for hash functions */
  /* table initialization omitted */

/* Windowed aggregate that maintains the hash−tables */
WINDOW AGGREGATE MaintainHashes(k Int):Int {
  /*an aggregate that updates a certain hash entry */
  AGGREGATE updateCnt(k Int, h Int, ah Int, bh Int, val Int):Int {
    INITIALIZE: ITERATE: {
      UPDATE hash_tables SET cnt = cnt+val
       WHERE index1 = ((ah*k+bh)%31)%4 AND index2 = h }
  };
  INITIALIZE: ITERATE: { /* new item entering the window */
    SELECT updateCnt(k, h, ah, bh, 1) FROM hs
  }
  EXPIRE: { /* item expiring */
    SELECT updateCnt(k, h, ah, bh, −1)FROM hs }
};
    /* Calling the UDA just defined*/
SELECT MaintainHashes(item) OVER (ROWS 29 PRECEDING)
FROM items;
```

5 Mining Data Streams

Data stream mining represents an important area of current research, and the topic of many recent papers, which primarily focus on devising mining algorithms that are fast and light enough to be executed continuously and produce real-time or quasi real-time responses. However, online data mining represents such a difficult issue for DSMS that no system before Stream Mill [40] has claimed success in this important application. Many of the problems facing DSMS are similar to those of DBMSs that in mid-1990s where unable to extend the success of SQL on OLAP applications to data mining applications. Indeed performing data mining tasks through the DBMS-supported constructs and functions was exceedingly difficult [41], whereby in a visionary 1996 paper [42], Imielinski and Mannila called a major research effort to produce quantum leap in the functionality and usability of DBMSs, whereby mining queries can be supported with the same ease of use as other relational queries are now supported. The notion of "Inductive DBMS" was thus born, which inspired much research [43], while vendors have been working on providing some data mining functionality as part of their DBMS [44].

The Stream Mill DSMS supports a powerful data stream mining workbench called SMM which is open and extensible. The first ingredient of SMM is a library of powerful data stream mining methods defined as window UDAs. New methods can be defined using ESL, or procedural languages; in either case the definition follows the standard INITIALIZE, ITERATE, TERMINATE, and EXPIRE templates of aggregates previously described. For the analysts and other users who want to work at higher level of abstraction, SMM support mining models [40, 45]. Mining methods and models are discussed next.

5.1 Density-Based Clustering (DBScan)

DBScan represents a popular clustering algorithm that can be successfully applied to mining and monitoring data streams [46]. Let us assume we have a stream of two-dimensional data, where more than **minPts** points occurring in close proximity (i.e., at distance less than **eps**) of each other are assigned to the same cluster, while sparse points are instead classified as outliers. To monitor changes in the incoming stream of two-dimensional data, we employ DBScan algorithm as follows: (i) partition the stream into blocks containing the same number of tuples, (ii) cluster the data in each block, and (iii) monitor the appearance/disappearance of new/old clusters and changes in cluster population between successive blocks. The first two tasks are accomplished by the following ESL statement that invokes the **dbscan** aggregate on input data stream **Stream_of_Points(Xvalue, Yvalue, TimeStamp)**:

```
/*call dbscan with minPts = 10 and eps = 50 */
SELECT dbscan(Xvalue, Yvalue, 0, 10, 50)
        OVER(ROWS 999 PRECEDING SLIDE 1000 )
FROM Stream_of_Points
```

Here 10 and 50 are the example values we assign to two important parameters for the DBScan Algorithm, minPts and eps, respectively. The third argument is for book-keeping purposes. Observe that since the size of the slide is the same as that of the window, this is a tumble. Therefore the Stream Mill system will use the base definition of DBScan, shown below, independent of whether a window version is available or not.

Given the two parameters **eps** and **minPts**, the DBScan algorithm works as follows: pick an arbitrary point **p** and find its neighbors (points that are less than eps distance away). If **p** has more than **minPts** neighbors then form a cluster and call DBScan on all its neighbors recursively. If **p** does not have more than **minPts** neighbors then move to other un-clustered points in the database. Note, this can be viewed as a depth-first search.

```
AGGREGATE dbscan(iX Real, iY Real, Flag Int, minPt Int, eps Int): Int
{       TABLE closepnts(X2 real, Y2 real, C2 Int) MEMORY;
        INITIALIZE: ITERATE: {
          /* Find neighbors of the given point */
```

```
        INSERT INTO CLOSEPNTS SELECT X1, Y1, C1 FROM points
          WHERE sqrt((X1-iX)*(X1-iX) + (Y1-iY)*(Y1-iY)) < eps;
        /* If there are more than minPt neighbors, form a cluster */
        UPDATE clusterno SET Cno= Cno+1  /* new cluster number*/
          WHERE Flag=0 AND SQLCODE=0 /* A new cluster */
            AND minPt < (SELECT count(C2) FROM closepnts);
        /* Assign these neighboring points to this cluster */
        UPDATE points SET C1 = (SELECT Cno FROM clusterno)
          WHERE points.C1=0 AND
            EXISTS (SELECT S.X1 FROM closepnts AS S
              WHERE points.X1=S.X2 AND points.Y1=S.Y2 )
              AND minPt < (SELECT count(C2) FROM closepnts);
        /* Call dbscan recursively */
        SELECT dbscan(X2, Y2, 1, minPt, eps)
          FROM  closepnts, points
          WHERE X1 = X2 AND Y1=Y2;
        DELETE FROM closepnts;
    }
};      /*end dbscan*/
```

5.2 *Mining Data Streams with Concept Drift*

Since streaming data is characterized by time-changing concepts, a basic challenge faced by data mining algorithms is to model and capture the time-evolving trends and patterns in the streams, and make time-critical predictions. One approach is to incrementally maintain a model for the time-changing data. In this case the model is learned from data in the most recent window. This approach has several weak points. First, given that data arrives at a high speed, incremental model maintenance is usually a costly task, especially for learning methods such as the decision tree algorithm, which are known to be unstable. Second, models trained from the data in a window may not be optimal. If the window is too large, it may contain concept drifts; if it is too small, it may result in over-fitting.

A more effective approach consists in using an ensemble based model whereby we partition the stream into fixed size data chunks and learn a model from each chunk. We combine models learned from data chunks, whose class distribution is similar to the most recent training data, to be our stream classifier (as shown in Fig. 1). This approach reduces classification error in the concept-drifting environment. We use ESL to implement this approach [47] effectively, which takes full advantage of off-the-shelf classifier packages and other procedural routines.

The seemingly complex solution described above can be implemented in ESL in a very succinct way. We assume each data record in the stream is in the form of $(\mathbf{a_1}, \ldots, \mathbf{a_n}, \mathbf{L})$, where $\mathbf{a_i}, \ldots, \mathbf{a_n}$ are attribute values, and $\mathbf{L}$ is the class label. If $\mathbf{L} =$

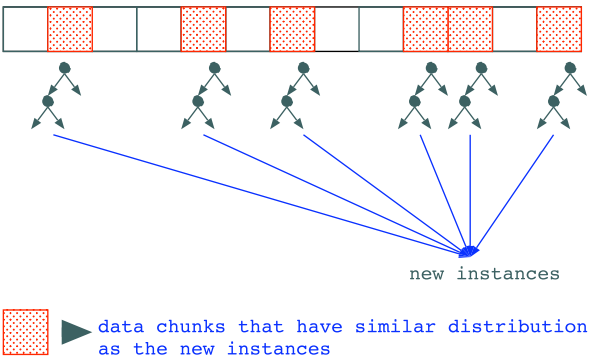

Fig. 1 Mining streams with concept-drift

TBA then it is a testing example, otherwise it is a training example. In Example 12, we express the algorithm in one SQL statement.

Here, we call UDA ClassifyStream with keyword SLIDE, which implements data partitioning on the stream, via tumbles of size 1000. We assume classifiers, together with their weights, are stored in a table called ensemble. In UDA **classifystream**, we use classifiers in the ensemble whose weights are above a given threshold to classify each test example, where Classify is a UDA for classifying static data [11]. Once we reach the end of a data partition, we learn a new classifier from the training data in the partition, and we reset the weight of each classifier in the ensemble proportional to its accuracy in classifying the most recent training data. The freshly weighted classifiers will then be used to classify data in the next partition.

Example 12 A Terse Expression for Complex Classifier Ensembles

```
SELECT ClassifyStream(S.*)
       OVER (ROWS 999 PRECEDING SLIDE 1000)
FROM stream AS S;
```

Example 13 UDA **classifystream**

```
AGGREGATE ClassifyStream(a1, ..., an, L) : Int
{    TABLE temp(a1, ..., an, L);
     INITIALIZE : ITERATE : {
        INSERT INTO RETURN
           SELECT sum(E.Classify(a1, ..., an) × E.weight) /
                  sum(E.weight)
           FROM ensemble AS E
           WHERE L = TBA AND E.weight ≥ threshold;
        INSERT INTO TEMP VALUES (a1, ..., an, L);
     }
     TERMINATE : {
        INSERT INTO ensemble
```

```
        SELECT learn(T.*) FROM TEMP AS T
        WHERE T.L <> TBA;
    UPDATE ensemble AS E SET E.weight =
        (SELECT 1-avg(|E.Classify(T.*)-T.L|)
         FROM TEMP AS T
         WHERE T.L<>TBA);
  }
}
```

5.3 Mining Models

The integration of mining methods into SMM is made simple via the Mining Model Definition Language which support the declaration of mining models [40]. Each mining model instance defines (i) which mining UDAs will be used in the task, (ii) the parameter values and ancillary information they will use, and (iii) the flow of stream data between these methods [40, 45].

Seldom *flows* need to be specified for complicated mining tasks. Consider a more advanced mining method such as an ensemble based weighted bagging (EBWB) [47], which is supported by SMM to improve the accuracy of classifiers in the presence of concept drifts and shifts. With EBWB, instead of maintaining a single classifier, the user maintains several small classifiers, whose classification in combined later using some kind of weighted voting. This approach assures a better adaptation in the presence of concept-shift and concept-drift, since new classifiers can be continuously trained based on the latest statistics, while older or inaccurate classifiers can be retired. Note that specifying the various steps required for weighted bagging represents a daunting task for analysts and less experienced users. Therefore, MMDL supports specification of one or more mining *flows* within the mining model definition. These complex mining processes only have to be specified once during model definition and can be reused by all users. *Flows* have been essential in definition of many built-in mining methods, such as SWIM for association rule mining [48], in SMM [40]. At the best of our knowledge SMM's ability to support data stream mining algorithms is unique among DSMS and CEP systems. On the other hand, systems such as MOA [49] provide a flexible and user-friendly environment for evaluating algorithms for data stream mining and for the incremental mining of data sets; however such systems are not DSMS designed to support QoS for continuous queries over extended periods of time.

6 The Stream Mill System

The architecture of the Stream Mill system consists of a single server and multiple clients.

The Client

Users interact with the server through the query editor—marked as α in Fig. 2. The query editor allows the user to perform the following tasks: logging in and out of the system, defining streams, queries, aggregates, starting and stopping queries, etc. Results of these tasks are shown in the query editor's status pane, by default. The client also provides a set of GUI modules to display the workflow and the results of the continuous queries in a graphical form, e.g., β in Fig. 2. Several performance meters (marked as γ) are also at hand to continuously monitor traffic, memory utilization, queue length, and related measures of server performance.

The Server

The bulk of Stream Mill R&D efforts focused on the Server, which supports the following functional modules:

Query Compiler/Optimizer

The compiler is responsible for parsing and compiling continuous queries and generating/modifying the query graph that describes how continuous queries are implemented by operators that take tuples from their input buffers and push them into their output buffers (in cooperation with the Buffer Manager). These operators are implemented as C/C++ functions compiled into dynamic libraries, which are then invoked by the Execution Scheduler. After careful optimizations, natively defined UDAs on the average execute nearly as well (a 30 % slowdown) as UDAs externally defined in C++ and better than those defined in Java

Buffer Manager

The Buffer Manager is responsible for managing the stream tuples as in-memory queues. Most tuples are processed and removed from these buffers as quickly as possible to free space and to reduce latency. However, when windows are used in the query, tuples must be retained main memory for extended period of time. This is the task of the *Window Manager*, which is also responsible for supporting delta-maintenance constructs for these windows, window sharing, and suitable paging policy.

Execution Scheduler

The Execution Scheduler is responsible for deciding which operator from the query graph executes next and how many input tuples it will consume. This decision reflects the optimization criterion selected, which in turn reflects priorities specified

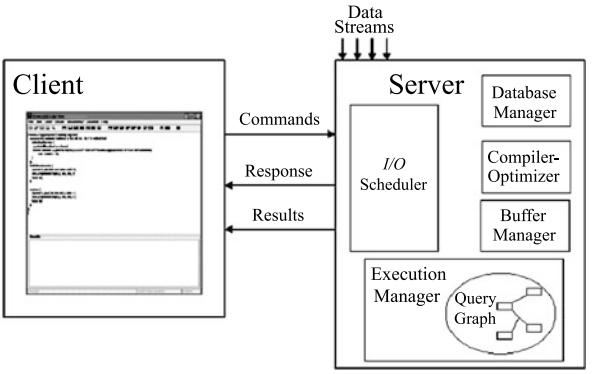

Fig. 2 The stream mill system architecture

by the user and also the load conditions currently experienced in the system. For instance, a low-memory condition might force the scheduler to change from a scheduling policy that minimizes response time to one that minimizes memory [50]. This change might require a modification (e.g., partitioning) of the query graph; Stream Mill is capable of performing this adjustment very quickly, efficiently, and without stopping the execution of the continuous queries [51]. Stream Mill's flexible execution model also supports on demand-generation of timestamp to minimize idle-waiting in operators such as union and joins [51].

Other important modules which are part of the system include the *I/O Scheduler*, which is responsible for managing incoming and departing data streams, and the interaction between the server and the outside world, including the Stream Mill Client. The *Database Manager,* based on *ATLaS* and *Berkeley DB*, is used to support database queries and spanning applications. Various extensions and improvements being added include new data stream mining algorithms, load shedding extensions, and better GUI primitives.

7 Related Work

Data Stream Management Systems (DSMSs) represent a vibrant area of current research comprising several subareas. Because of space limitations and the availability of authoritative surveys [5, 52], we will here focus on previous works that are most relevant to ESL.

The Tapestry project [53, 54] that was the first to discuss 'queries that run continuously over a growing database' with append-only relations as the basic data model for data streams. The append-only data model was then adopted in many projects, including Tribeca [55], and Telegraph [56]. Likewise, OpenCQ [57] and Niagara Systems [58] are designed to support continuous queries for monitoring web sites, and the Chronicle data model uses append-only ordered sets of tuples (chronicles) [59].

With respect to languages for continuous queries, the use of SQL and its dialects is predominant not only for DSMS of relational DBMS lineage, but also for the

various systems of mongrel lineages known as CEP systems.[7] In reality, however, the apparent popularity of SQL is restricted by many exceptions and limitations. For instance, extensions to C++ are used in Hancock [2], while systems focusing on streaming XML data use XQuery or Xpath [8, 60]. Finally, CEP systems tend to support SQL only as a tool of convenience for simple applications, while they rely on some Java-based language for more serious applications and system extensions. Even in DSMS with a relational database lineage, we find variations and limitations. For instance, Tribeca relies on operators adapted from relational algebra [55], while active database rules are used in OpenCQ [57]. Furthermore, the influential Aurora/Borealis project [61] focuses on providing an attractive graphical interface to define a network of continuous query operators, where the user can then request an equivalent SQL program to be produced from this.

Another influential DSMS project is STREAM; this system and its Continuous Query Language (CQL) [14] features several syntactic variations from the SQL:2003 standards, and from the append-only model, by proposing an approach based on database queries over continuously sliding windows.

Unlike many other DSMS projects, however, the Stream Mill project seeks to preserve the syntax and semantics SQL standards as far as possible. In this respect, our project is similar to the very influential Gigascope [15] project, which has also adopted the append-only model as ESL does. But unlike Gigascope, which was designed primarily for network analysis and management, ESL strives to serve a much wider range of applications, including applications not supported by other DSMSs, such as pattern queries on relational and XML streams [25, 28], and data mining queries [40]. Thus, while Gigascope relies on SQL-2 style of aggregates, ESL adopts the SQL:2003 constructs for windowed aggregates; the same constructs are then applied to UDAs producing a compact language that is Turing complete on stored data and NB-complete on streaming data [6, 62].

8 Conclusion

A key contribution of ESL and Stream Mill is proving that rather limited extensions enable database query languages to support effectively a very wide range of data stream applications, by providing levels of expressive power and generality that match or surpass those of other query languages and systems proposed for data stream and publish/subscribe applications [1–3, 8, 9]. The merits of ESL extensions are supported by theoretical results [6, 62] and demonstrated by important applications that are beyond the reach of other DSMS, including continuous data mining queries [40], sequence queries on the nested-word and the XML model [24, 25, 28], and algorithms for synopsis maintenance [36]. This significant leap in power and generality has been achieved while preserving the basic append-only-table semantics for data streams and minimizing extensions w.r.t. SQL:2003 standards—as

[7] http://en.wikipedia.org/wiki/Complex_event_processing.

needed to facilitate the writing of applications that span both databases and data streams.

The Stream Mill prototype is now fully operational and supports (i) continuous queries on data streams [63], (ii) ad hoc queries on database tables, and (iii) ad hoc queries on table-like concrete views defined on data streams. More information on (iii), time-series queries, and XQuery on SAX in Stream Mill is available from the project web site [13].

Acknowledgements Thanks are due to Yijian Bai, Yannei Law, Stefano Emiliozzi, Shu Man Li, Vincenzo Russo, and Xin Zhou for their many contributions to the system and its enabling technology.

References

1. D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S.Z. Aurora, A new model and architecture for data stream management. VLDB J. **12**(2), 120–139 (2003)
2. C. Cortes, K. Fisher, D. Pregibon, A. Rogers, Hancock: a language for extracting signatures from data streams, in *SIGKDD* (2000), pp. 9–17
3. P. Felber, P. Eugster, R. Guerraoui, A. Kermarrec, The many faces of publish/subscribe. ACM Comput. Surv. **35**(2), 114–131 (2003)
4. ISO/IEC. Database languages—SQL, ISO/IEC 9075-*:2003 (2003)
5. B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom, Models and issues in data stream systems, in *PODS* (2002), pp. 1–16
6. Y.-N. Law, H. Wang, C. Zaniolo, Data models and query language for data streams, in *VLDB* (2004), pp. 492–503
7. I. Information Technologies, Illustra user's guide, in *1111 Broadway, Suite 2000*, Oakland, CA (1994)
8. D. Florescu, C. Hillery, D. Kossmann et al., The BEA/XQRL streaming xquery processor. VLDB J. **13**(3), 294–315 (2004)
9. Oracle. Oracle9i application developer's guide advanced queuing. Oracle, Redwood Shores, CA, USA (2002)
10. H. Wang, C. Zaniolo, Using SQL to build new aggregates and extenders for object-relational systems, in *VLDB* (2000), pp. 166–175
11. H. Wang, C. Zaniolo, Atlas: a native extension of sql for data minining, in *Proceedings of Third SIAM Int. Conference on Data Mining* (2003), pp. 130–141
12. J. Li, D. Maier, K. Tufte, V. Papadimos, P.A. Tucker, Semantics and evaluation techniques for window aggregates in data streams, in *SIGMOD Conference* (2005), pp. 311–322
13. Stream mill home. http://wis.cs.ucla.edu/stream-mill
14. A. Arasu, S. Babu, J. Widom, Cql: a language for continuous queries over streams and relations, in *DBPL* (2003), pp. 1–19
15. C. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, O. Spatscheck, Gigascope: high performance network monitoring with an sql interface, in *SIGMOD* (ACM, New York, 2002), p. 623
16. R. Sadri, C. Zaniolo, A. Zarkesh, J. Adibi, Optimization of sequence queries in database systems, in *PODS* (2001)
17. R. Sadri, C. Zaniolo, A.M. Zarkesh, J. Adibi, Expressing and optimizing sequence queries in database systems. ACM Trans. Database Syst. **29**(2), 282–318 (2004)
18. F. Zemke, A. Witkowski, M. Cherniak, L. Colby, Pattern matching in sequences of rows, in *Sql Change Proposal* (2007). http://www.sqlsnippets.com/en/topic-12162.html

19. N. Dindar, B. Güç, P. Lau, A. Ozal, M. Soner, N. Tatbul, Dejavu: declarative pattern matching over live and archived streams of events, in *SIGMOD Conference* (2009), pp. 1023–1026
20. E. Wu, Y. Diao, S. Rizvi, High-performance complex event processing over streams, in *SIGMOD Conference* (2006), pp. 407–418
21. D. Gyllstrom, J. Agrawal, Y. Diao, N. Immerman, On supporting kleene closure over event streams, in *ICDE* (2008), pp. 1391–1393
22. A.J. Demers et al., Cayuga: a high-performance event processing engine, in *SIGMOD Conference* (2007), pp. 1100–1102
23. R.S. Barga et al., Consistent streaming through time: a vision for event stream processing, in *CIDR* (2007), pp. 363–374
24. B. Mozafari, K. Zeng, C. Zaniolo, K*SQL: a unifying engine for sequence patterns and XML, in *SIGMOD Conference–Demo Track* (2010), pp. 1143–1146
25. B. Mozafari, K. Zeng, C. Zaniolo, From regular expressions to nested words: unifying languages and query execution for relational and XML sequences. Proc. VLDB Endow. **3**(1), 150–161 (2010)
26. R. Alur, P. Madhusudan, Adding nesting structure to words, in *Developments in Language Theory* (2006)
27. R. Alur, P. Madhusudan, Visibly pushdown languages, in *STOC* (2004), pp. 202–211
28. X. Zhou, H. Thakkar, C. Zaniolo, Unifying the processing of XML streams and relational data streams, in *ICDE* (2006), p. 50
29. U. Srivastava, J. Widom, Memory-limited execution of windowed stream joins, in *VLDB* (2004), pp. 324–335
30. Y. Bai, H. Thakkar, C. Luo, H. Wang, C. Zaniolo, A data stream language and system designed for power and flexibility, in *CIKM* (2006), pp. 337–346
31. L. Golab, M. Tamer Özsu, Update-pattern-aware modeling and processing of continuous queries, in *ACM SIGMOD Conference* (2005), pp. 658–669
32. B. Mozafari, C. Zaniolo, Optimal load shedding with aggregates and mining queries, in *ICDE* (2010), pp. 76–88
33. Y.-N. Law, C. Zaniolo, Improving the accuracy of continuous aggregates and mining queries on data streams under load shedding. Int. J. Bus. Intell. Data Min. **3**(1), 99–117 (2008)
34. M. Datar, A. Gionis, P. Indyk, R. Motwani, Maintaining stream statistics over sliding windows: (extended abstract), in *Proceedings of the Thirteenth Annual ACM–SIAM Symposium on Discrete Algorithms* (2002), pp. 635–644
35. C. Aggarwal, *Data Streams: Models and Algorithms* (Springer, Berlin, 2007)
36. H. Mousavi, C. Zaniolo, Fast and accurate computation of equi-depth histograms over data streams, in *EDBT* (2011), pp. 69–80
37. C. Jin, W. Qian, C. Sha, J.X. Yu, A. Zhou, Dynamically maintaining frequent items over a data stream, in *Proceedings of the 12th ACM Conference on Information and Knowledge Management (CIKM)* (2003)
38. M. Charikar, K. Chen, M. Farach-Colton, Finding frequent items in data streams, in *International Colloquium on Automata, Languages, and Programming (ICALP)* (2000), pp. 508–515
39. G. Cormode, S. Muthukrishnan, What's hot and what's not: tracking most frequent items dynamically, in *PODS* (2003), pp. 296–306
40. H. Thakkar, N. Laptev, H. Mousavi, B. Mozafari, V. Russo, S.M.M. Carlo Zaniolo, A data stream management system for knowledge discovery, in *ICDE* (2011), pp. 757–768
41. S. Sarawagi, S. Thomas, R. Agrawal, Integrating association rule mining with relational database systems: alternatives and implications, in *SIGMOD* (1998)
42. T. Imielinski, H. Mannila, A database perspective on knowledge discovery. Commun. ACM **39**(11), 58–64 (1996)
43. C. Zaniolo, Mining databases and data streams with query languages and rules—invited paper, in *KDID 2005: Knowledge Discovery in Inductive Databases, 4th International Workshop*. Lecture Notes in Computer Science, vol. 3933 (Springer, Berlin, 2006), pp. 24–37
44. Z. Tang, J. Maclennan, P.P. Kim, Building data mining solutions with OLE DB for DM and XML for analysis. SIGMOD Rec. **34**(2), 80–85 (2005)

45. H. Thakkar, B. Mozafari, C. Zaniolo, Designing an inductive data stream management system: the stream Mill experience, in *SSPS* (2008), pp. 79–88
46. H.-P. Kriegel, M. Ester, J. Sander, X. Xu, A density-based algorithm for discovering clusters in large spatial databases with noise, in *KDD* (1996), pp. 226–231
47. H. Wang Wei Fan, P.S. Yu, J. Han, Mining concept-drifting data streams using ensemble classifiers, in *KDD* (2003), pp. 226–235
48. B. Mozafari, H. Thakkar, C. Zaniolo, Verifying and mining frequent patterns from large windows over data streams, in *ICDE* (2008), pp. 179–188
49. A. Bifet, G. Holmes, B. Pfahringer, P. Kranen, H. Kremer, T. Jansen, T. Seidl, Moa: massive online analysis, a framework for stream classification and clustering. J. Mach. Learn. Res. **11**, 44–50 (2010)
50. Y. Bai, C. Zaniolo, Minimizing latency and memory in DSMS: a unified approach to quasi-optimal scheduling, in *SSPS* (2008), pp. 58–67
51. Y. Bai, H. Thakkar, H. Wang, C. Zaniolo, Optimizing timestamp management in data stream management systems, in *ICDE* (2007), pp. 1334–1338
52. L. Golab, M. Tamer Özsu, Issues in data stream management. ACM SIGMOD Rec. **32**(2), 5–14 (2003)
53. D. Barbara, The characterization of continuous queries. Int. J. Coop. Inf. Syst. **8**(4), 295–323 (1999)
54. D.B. Terry, D. Goldberg, D.A. Nichols, B.M. Oki, Continuous queries over append-only databases, in *SIGMOD Conference* (1992), pp. 321–330
55. M. Sullivan, Tribeca: a stream database manager for network traffic analysis, in *VLDB* (1996), p. 594
56. S. Chandrasekaran et al., TelegraphCQ: continuous dataflow processing for an uncertain world, in *CIDR* (2003)
57. L. Liu, C. Pu, W. Tang, Continual queries for Internet scale event-driven information delivery. IEEE Trans. Knowl. Data Eng. **11**(4), 583–590 (1999)
58. J. Chen, D.J. DeWitt, F. Tian, Y. Wang, NiagaraCQ: a scalable continuous query system for Internet databases, in *SIGMOD* (2000), pp. 379–390
59. H. Jagadish, I. Mumick, A. Silberschatz, View maintenance issues for the chronicle data model, in *PODS* (1995), pp. 113–124
60. A. Kumar Gupta, D. Suciu, Stream processing of xpath queries with predicates, in *SIGMOD Conference* (2003), pp. 419–430
61. D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, S. Zdonik, Monitoring streams—a new class of data management applications, in *VLDB*, Hong Kong, China (2002)
62. Y.-N. Law, H. Wang, C. Zaniolo, Relational languages and data models for continuous queries on sequences and data streams. ACM Trans. Database Syst. **36**, 8 (2011)
63. C. Luo, H. Thakkar, H. Wang, C. Zaniolo, A native extension of SQL for mining data streams, in *ACM SIGMOD Conference 2005* (2005), pp. 873–875