ESL: a Very Powerful SQL-Compliant Data Stream Language

Yijian Bai[1] Chang R. Luo[1] Hetal Thakkar[1] Haixun Wang[2] Carlo Zaniolo[1]

Computer Science Dept., UCLA[1]
Los Angeles, CA 90095
{bai,lc,hthakkar,zaniolo}@cs.ucla.edu

IBM T. J. Watson Research[2]
Hawthorne, NY 10532
haixun@us.ibm.com

Abstract

Compliance with SQL standards is very desirable for a data stream query language because of practical considerations, and it is also very beneficial for applications that span both data streams and data bases. However, SQL suffers from expressive power impairments that, on streaming data, are even more serious than those it suffers on stored data. Our Expressive Stream Language (ESL) solves these problems, and achieves power, flexibility, and adherence to SQL:2003 standards by using: (i) table expressions and concrete views on data streams, (ii) non-blocking aggregates (UDAs), and (iii) efficient delta-based maintenance for UDAs on windows. ESL is fully supported in the UCLA Stream Mill DSMS and has proven very effective on a wide spectrum of applications that include approximate computations, data stream mining, time-series queries, and XML streams.

1 Introduction

Data Stream Management Systems (DSMSs) and their languages represent a vibrant and exciting area of database research [9, 16]. However the requirements and applications for data received on the wire are so different from those for data stored on disk, the view that this burgeoning research area naturally belongs to databases is not commonly accepted outside our field. This incertitude also mires database vendors: while some have moved to add support for publish/subscribe and memory queues [3] into their DBMS, others have chosen to add these services to, say, web services [1]. A key objective of the ESL/Stream Mill Project is to settle these issues by showing that (i) standard SQL is all (or nearly all) that is needed to express continuous and ad hoc queries on data streams, and (ii) this approach is effective on a very wide range of applications—e.g., XML stream processing and stream mining. Achieving these two goals will go a long way toward enticing database vendors and convincing skeptics on the superiority of a database-oriented approach. Users will also greatly benefit from these advances, because many applications span both data streams and databases [12]. The users writing these applications will then be able to employ the same language on both streaming data and stored data, rather than having to learn two languages and deal with their impedance mismatch.

Therefore, ESL pursues the parallel objectives of (i) minimizing the syntactic and semantic differences from SQL standards (i.e., SQL: 2003), and (ii) supporting a very wide range of applications, by bringing SQL to new levels of expressive power and flexibility. The difficulty of achieving these objectives is underscored by the long-known expressive power limitations of SQL on stored data, and by recent research results showing that the problem is even more serious for continuous queries on data streams [20].

The main negative result presented in [20] comes from the fact that non-blocking queries are exactly monotonic queries. However, if we eliminate from SQL its non-monotonic operators (such as EXCEPT), we also lose some of its monotonic query expressibility. Thus SQL is not complete w.r.t. non-blocking queries (and neither is relational algebra) [20]. These limitations are exacerbated by the fact that the traditional remedy of embedding SQL queries in a procedural programming language (PL) (through the pull-based mechanism of cursors and get-next constructs) is no longer effective in the push-based environment of data streams. DSMS must operate in a push-oriented environment by continuously taking tuples from input buffers, and continuously pushing the query results to output buffers—without waiting for get-next requests from an embedding PL.

Fortunately, the following encouraging positive result was also proved in [20]: SQL becomes Turing-complete, and also complete w.r.t. non-blocking queries, once it is extended with user defined aggregates (UDA) [1]. In this paper, we build on this theoretical result, and turn UDAs into a tool of great practical significance in a broad range of data stream applications, by extending them with window constructs that had in the past been used for built-in aggregates

Proceedings of the 31st VLDB Conference,
Trondheim, Norway, 2005

[1]While UDAs are not part of the official SQL:2003 standards, they hardly represent an extension, since they were included in early SQL3 draft, and are supported in many commercial DBMS.

but not for UDAs.

Because of space limitations and the availability of authoritative surveys [9, 16], we will not discuss here previous projects, except for observing that they are not focused on the two objectives of (i) compliance with SQL standards, and (ii) generality through expressive power, with the same ardor as ESL. For instance Aurora/Borealis, rather than using SQL, provides an operator-based graphical user interface for entering continuous queries. The Stanford Continuous Query Language (CQL) [6] is based on SQL—but compliance with standards is not a key objective driving the design of the language. In particular, CQL allows window construct in the FROM clause to be used in the specification of joins, while in SQL windows are only allowed as aggregate modifiers in the SELECT clause. Similar constructs having different syntax and semantics in CQL and SQL complicates the task of the programmer writing spanning applications. The CQL constructs are different enough from standard SQL to require a new formal definition for its semantics [8]. Moreover, a somewhat arbitrary choice must be made between the different semantics proposed for joins involving windows. In ESL, we instead rely on standard syntax and semantics of SQL 2003, and rather than introducing new constructs for window joins, we express them through the existing constructs as discussed in this paper.

Short Overview

In the next section, we cover the use of SQL constructs to express simple continuous queries involving the application of the following operators on data streams:

1. Select, Project, and Aggregates (with or without window modifiers) on a single data stream,
2. Join of a data stream with a DB table, and
3. Union of two or more data streams.

These operators can be used to define continuous queries or to derive new data streams from existing ones in a view-like fashion.

ESL also supports ad hoc queries on stored database tables and on virtual tables derived from data streams via SQL:2003 constructs of concrete views and table functions. These are discussed in Section 3, where we also show how to express the join of a stream with a window on another stream, using these constructs.

UDAs, which represent the cornerstone of ESL query power, are discussed in Section 4, which provides a simple syntactic characterization of non-blocking UDAs versus blocking UDAs. The former can be applied directly on data streams, while the latter can only be applied over windows (as in SQL OLAP functions).

The optimization of window UDAs is presented in Section 5: our approach is based on delta-maintenance techniques that are effective for different kinds of windows and aggregates.

In Section 6, we show that, because of its query power, ESL can concisely and efficiently express applications, such as data stream mining, sequence queries, and approximate computations, which would be impossible or difficult to handle in other query languages.

2 Continuous Queries on Data Streams

ESL treats data streams as unbounded ordered sequences of tuples: this is consistent with the 'append only table' model commonly used by data stream systems [11, 9, 16, 20]. In the ESL system, each data stream is imported from an external wrapper via the (mandatory) SOURCE clause in its CREATE STREAM declaration. This declaration also specifies the type of timestamp associated with the stream. ESL supports the following three types of timestamps: (i) *external timestamps*, (ii) *internal timestamps*, and (iii) *latent timestamps*.

External timestamps are values that are already contained in the arriving tuples—typically, placed there by the application producing the data; in this case, all that is needed, in the data stream declaration, is to identify the column containing such timestamps using the order-by clause. For instance, the data stream **OpenAuction** in Example 1, below, is declared as having **start_time** as its external timestamp.

The **ClosedAuction** stream in Example 1, is instead assigned an internal timestamp: internal timestamps are generated when the ESL system receives the tuples from the wrappers and they are stored in a new column called **current_time** - a reserved name used only to denote internal timestamps. Internal timestamps and external timestamps will be called *explicit*: ESL operators treat all explicit timestamps in the same way, no matter how they were generated. Explicitly timestamped streams are always ordered by increasing timestamp values. In Section 2.3, we discuss how this assumption is enforced by a special treatment of out-of-order records.

ESL also supports data streams, such as **Bid** in Example 1, where no explicit timestamp is actually stored in the tuples (a fact denoted by the absence of ORDER BY from their declarations). However, timestamps values (consistent with tuple order in the stream) are dynamically generated for these tuples whenever they are used in operators that have a semantics based on timestamps. Therefore, we refer to these data streams as having *latent* timestamps[2].

Example 1 *Declaring Streams in ESL*

```
CREATE STREAM OpenAuction ( /* Stream of auction openings */
            itemID int /* id of the item being auctioned.*/,
            sellerID char(10) /* seller of the item being auctioned.*/,
            start_price real /* starting price of the item */,
            start_time timestamp /* time when the auction started */)
    ORDER BY start_time; /* external timestamps */
    SOURCE 'port4445';
CREATE STREAM ClosedAuction(/*Stream of auction closings */
            itemID int /* id of the item in this auction. */,
            buyerID char(10) /* buyer of this item.*/)
            final_price real /* final price of the item */,
            current_time timestamp /*internal timestamps*/)
    ORDER BY current_time; /* internal timestamp */
    SOURCE 'port4446';
CREATE STREAM Bid( /* Bid: Stream of bidding.*/
    itemID int /* the item being bid for*/,
```

[2]In summary, external timestamps are generated by the external producer of the data, internal timestamps are generated eagerly as they enter the DSMS, and latent timestamps are produced lazily as needed for processing the tuples.

```
    bid_price real /* bid price */,
    bidderID char(10) /* id of the bidder*/,
    bid_time timestamp /* time when bid was registered */)
    SOURCE 'port4447';
```

Example 1 uses wrappers that are created automatically by the system for each port used in the program. Thus, for port '4446' the system creates a file named **'port4446'** containing the code that 'wraps' data coming from that port—with data items and records, respectively, separated by commas, and end-of-line characters. Rather than using these defaults, users can also easily create their own wrappers.

2.1 Single Stream Transducers

ESL only allows *one* data stream in FROM clause of the query; this is a restriction that ensures simpler syntax and semantics, without impairing the power of the language, as we shall see later. For instance, to continuously send to the user all auctions where the asking price is above 1000, we can write:

Example 2 *Performing Selection Operations on Streams*

```
    SELECT itemID, sellerID, start_price, start_time
    FROM OpenAuction WHERE start_price > 1000
```

Semantics. The clause 'ORDER BY **start_time**' can also be added to this query, without changing its meaning, since the tuples are always produced by the increasing values of timestamps. Therefore, consider the query in Example 2, after the addition of 'ORDER BY **start_time**': the semantics of this query in ESL is exactly the same as in SQL. Indeed, the ordered list of tuples produced by ESL up to time t is exactly the same as that produced by SQL on table containing the list of **OpenAuction** tuples that have arrived up to time t. Therefore, in ESL, the semantics of continuous queries on data streams can be simply defined by reducing them to that of equivalent SQL:2003 queries on database tables.

ESL also supports the derivation of one stream from another through a CREATE STREAM mechanism that can be viewed as similar to the CREATE VIEW mechanism in SQL. For instance, Example 3, below,

Example 3 *Deriving a New Streams*

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

defines a data stream that is basically the same as that delivered to the user by Example 2. However, instead of being delivered to the user, **expensiveItems** is now a data stream that can be used by other queries, as in Example 4.

Example 4 *Sending the Results of a Continuous Query to the Output*

```
    SELECT itemID, start_price, start_time
    FROM expensiveItems WHERE sellerID= 'JA9248'
```

Aggregates

Aggregates are the final construct that can be applied to an individual data stream (i.e., via an ESL statement that has only one data stream in its FROM clause). ESL indeed supports very powerful user-defined aggregates (UDAs) that make the language very expressive and extensible [20].

Example 5 shows the invocation of a UDA called **decay_online_avg** that computes the exponential decay of the closing values of auctions. Since blocking aggregates are not allowed on data stream, the ESL compiler also checks that **decay_online_avg** is a non-blocking UDA—a property that is easily inferrable from the syntactic structure of the UDA definition, as discussed in Section 4.

Most aggregates, including the traditional SQL:2 aggregates, are blocking and can be applied to data streams via windows only. ESL uses the standard SQL:2003 syntax of OLAP functions for such window aggregates, whereby the window specification is appended to the aggregate using the OVER clause [31]. For instance, Example 6, below, shows the use of an unlimited window, whereby the **min** returns the lowest start price seen so far.

Example 5 *The Recent Average of the Closing Bids*

```
    SELECT decay_online_avg(final_price)
    FROM ClosedAuction
```

Example 6 *The Smallest Asking Price, For Each Seller*

```
    SELECT itemID, sellerID, start_price, min(start_price)
            OVER(PARTITION BY SellerID
                    RANGE UNLIMITED PRECEDING) AS Price
    FROM OpenAuction
```

In this example, we use an unlimited window, which basically returns the cumulative min so far. ESL supports both logical windows (i.e., time-based) and physical windows (i.e., count-based), and the optional partition-by clause whereby the incoming stream can be partitioned into multiple streams[3]. The only departure from SQL:2003 supported in ESL, is the option of omitting the ORDER BY clause, since the output data stream is already known to be ordered by its timestamp. Thus, UDAs invoked without a window modifier will be called *base aggregates* as in Example 5, whereas UDAs invoked with a window modifier will be called *window aggregates* as in Example 6.

Both the min aggregate and its window version are supported in ESL as built-ins, and so are the other basic SQL:2 aggregates. But ESL also supports efficiently window constructs on arbitrary UDA—not just builtin ones. This feature provides much greater power and flexibility than those provided by other DSMS or commercial implementation of OLAP functions that only support window on builtin aggregates, and will be discussed in Section 5.

Continuous Queries Spanning Data Streams and DB Tables.

ESL also supports the join of a data stream with database relations. For instance, if we have the database table

sellerinfo(sellerID, ZipCode, City, State)

then the following query can be used to add the zipcode of the seller to the **expensiveItems** stream:

Example 7 *Joining a Data Stream with a Database Table*

```
    SELECT ZipCode, itemID, start_price, start_time
    FROM expensiveItems AS I, sellerInfo AS S
    WHERE I.sellerID= S.sellerID
```

[3]The standard GROUP BY construct is used in ESL to partition the input streams for base UDAs.

The previous example only uses one database table in its FROM clause, in general, any number of database tables can be used in the FROM clause of an ESL query. However, only one data stream can be included in the FROM clause—and will be listed first in our examples for clarity. Furthermore, any nested subqueries are supported in ESL but they can only use database tables in their FROM clause. Therefore, with only one data stream allowed in their FROM clauses, ESL queries can be viewed as a transducer that takes one input stream and returns one output stream. This simple model is amenable to simple semantics for queries involving one data stream and multiple database tables.

Semantics. In the previous sections, we have defined the semantics of continuous queries involving only data streams by prescribing that, the cumulative result produced by a continuous query up to time t should be the same as that produced by its equivalent SQL:2003 statements applied to the content of the input data streams up to time t. However, for queries spanning both data streams and DB tables, we must also consider changes in the content of tables, by using a snapshot semantics for database tables. Thus, in Example 7, a new tuple from **expensiveItems** arriving at time t must be joined with snapshot of the **sellerinfo** table at time t. This delta result is then appended to the current output to produce the cumulative result of the query at time t. Observe that when no database tables are involved, or the content of the database tables involved does not change with time, we obtain the same result as the cumulative semantics discussed, previously.

2.2 UNION

UNION is the only ESL operator that is directly applicable to multiple data streams. (ESL does not allow EXCEPT and INTERSECT to be applied on data streams; and the union of data streams and database tables is also not allowed.)

For instance, the query in Example 8, below, sort-merges the **OpenAuction** and the **ClosedAuction** information on **start_price** and **final_price** for each item:

Example 8 *Price History Query*

```
CREATE STREAM PriceRise(itemID, price, Time) AS
    SELECT itemID, start_price, start_time
    FROM OpenAuction
    UNION ALL
    SELECT itemID, final_price, current_time
    FROM ClosedAuction
```

The union of data streams with explicit timestamps produces a stream that is ordered by its timestamps. Thus, the ORDER BY **Time** in the following ESL statement is immaterial and can be eliminated (but is included here to show the equivalent SQL query):

```
SELECT * FROM PriceRise ORDER BY Time
```

At the implementation level, union is implemented by a sort-merge operation on the streams. We choose the tuple with the minimum timestamp from the input data streams if none of their buffers are empty. If the buffer of any stream is empty, we must wait for its next incoming tuple before we proceed, since that tuple may have a timestamp smaller than any unprocessed tuple of the other streams.

As in SQL, ESL supports UNION ALL which preserves duplicates, and UNION, which eliminates them. (Duplicate elimination is efficiently supported in the sort-merge, since duplicate tuples must also have identical timestamps.)

2.3 Latent Timestamps

Timestamps explicitly stored in tuples are expensive to support, because of the space they use in buffers, and the processing cost they impose on query operators. Therefore, ESL introduces the notion of latent timestamps to avoid these costs in the many applications where explicit timestamps are not actually needed.

For instance in Example 1, the stream **Bid** has latent timestamps, since it is declared without the order-by clause. Also, as shown in Examples 5 and 6 a data stream where its explicit timestamp is projected out becomes one with latent timestamps.

Latent timestamps are not carried along in the tuples as these flow from one operator to the next: they are instead generated dynamically as required by the query operators being executed. In terms of semantics, the key difference between explicit and latent timestamps pertains to when their values are actually materialized: latent timestamps are materialized just before the tuples are processed by each query operator. Moreover, the abstract semantics of query operators on data streams is independent of the particular type of timestamp the data streams have [4]. This is illustrated by Example 9, where a time-based window is applied to data stream **Bid** which has latent timestamps: this query reports the number of bids received in the last hour (60 minutes) for each item.

Example 9 *For Each Item Count the Number of Bids in the Last Hour (60 Minutes)*

```
CREATE STREAM HActivty
    SELECT itemID, count(bid_price)
        OVER(PARTITION BY itemID
            RANGE 60 MINUTES PRECEDING)
    FROM Bid
```

The meaning of this statement is that when a new tuple is added to the window, a timestamp value equal to the current system time is created and stored with the tuple in the window. This timestamp is consulted to decide when the window tuple has expired from the window. The (unnamed) timestamp column that was used to maintain the window in Example 9 is not included in the resulting stream **HActivty**, which therefore has latent timestamps. Later query operators on **HActivty** that require timestamps will then generate new values based on the current clock.

The fact that the generated value of the latent timestamp is not part of the output simplifies the operational semantics and the implementation of query operators on data streams with latent timestamps; in fact the actual generation of the timestamps can be omitted all together for simple query operators, which preserve the arrival order of the tuples. For instance, the need for a sort-merge operation to compute union is eliminated by the observation that tuples could be assigned timestamp values equal to the time at which they were taken from their input buffers and moved to the output. The actual order of tuples being produced by such a

[4]Thus, the abstract semantics of queries on data streams with latent timestamps follows directly from that of stream with explicit timestamps which we have defined in the previous sections.

union depends on the order in which they arrived at the input buffers, and also on load conditions— but the order of tuples for each individual stream is always preserved.

Out-of-Order Tuples

Latent data streams offer many benefits besides performance. They are particularly useful with out-of-order tuples in a stream. For each externally timestamped stream ESL also generates a latent stream containing the out of order tuples. For instance, the declaration of **OpenAction** in Example 1, generates the data stream **OpenAction_OutOfOrder** with latent timestamps. Stream **OpenAction_OutOfOrder** contains all the out-of-order tuples of **OpenAction**. The user is thus given the opportunity to readjust the timestamps of **OpenAction_OutOfOrder**, e.g., to merge them back into the original stream.

The next example illustrates how data streams with latent timestamp combined with UDAs can be effectively used to fix timestamps and out-of-order problems. Example 10 below uses a window of 2 minutes. As discussed in Section 5, **resort_bids** is a UDA that selects out of a set of tuples (in this case the tuples within a two minutes window) those that are minimal in their fourth column and return these tuples (UDAs in ESL can return whole tuples, not just single values).

Example 10 *Resorting to Handle Late Arrivals by 2 Minutes or Less*

```
CREATE STREAM OKBids
    SELECT resort_bids(itemID, bid_price, bidderID, bid_time)
                    OVER(RANGE 2 MINUTES PRECEDING)
    FROM Bid
```

This solution only works when the maximum delay is less than 2 minutes. The tuples that are more than 2 minutes late can be deleted or given new timestamps, see Section 5.

3 Ad Hoc Queries and Concrete Views on Data Streams

While they have not received much attention in previous DSMS, ad hoc queries on data streams can be very useful. For instance, a user might want to find the average price and the volume of a given stock during the last 20 minutes—without having to receive that information continuously. The difference between the two kind of queries is that continuous queries persist until they are turned off by the users, ad hoc queries are instead turned off as soon as they complete, and they remain off until they are re-issued by the user.

Therefore, along with traditional ad hoc queries on database tables ESL also supports ad-queries on data streams through the mechanism of *concrete views* and *table functions* that extract snapshots of tables from the data streams.

Table functions represent a very powerful SQL:2003 construct that can be used to transform practically any kind of data source into SQL tables [25]. Thus we will use them to extract tables from data streams. Consider the following ad hoc query:

Example 11 Concrete View: *Count the number of items offered for auction in the last 10 minutes.*

```
SELECT itemID, sellerID
          FROM TABLE(OpendAuction OVER
               (RANGE 10 MINUTES PRECEDING CURRENT))
```

The syntax TABLE(**function_call**) is the standard SQL:2003 syntax (that explicitly reminds the user that the object returned is of type table). However, ESL allows the more user-friendly syntax for **function_call**, where over is the name of the function and 10 MINUTES PRECEDING CURRENT define the arguments of the function. In spite of the use of similar keywords our table-function construct is semantically different from the OVER construct used as aggregate modifier in several ways, including the fact that PARTITION BY is not allowed here.

Upon receiving such ad hoc query form the user, a system must collect 10 minutes of data before an answer can be returned to the user. One way to solve this waiting problem is to create the concrete view in Example 12, followed by the ad-hoc query in Example 13, as below.

Example 12 Concrete View: *A table containing all the auctions for the last 10 minutes.*

```
CREATE TABLE high_priced AS SELECT itemID, sellerID
          FROM TABLE( OpendAuction OVER
               (RANGE 10 MINUTES PRECEDING CURRENT))
REFRESH IMMEDIATE
```

Example 13 Ad Hoc Query on the concrete view: *Count the number of such auctions.*

```
SELECT count(ItemID) FROM high_priced
```

Because of the REFRESH IMMEDIATE option, the system will continuously keep the view up to date; then the ad hoc queries on this view can be answered immediately. A wide range of ad hoc queries can be answered efficiently on the view of Example 12, and could thus justify its creation, whereas a much more specific concrete view should be used to support only the query of Example 13. Indeed what concrete views should be created to optimally support a given set of ad-hoc queries represents an interesting problem [17].

In summary, ESL uses the following simple rule: Queries that use a data stream in their FROM clause are continuous; queries that only use database tables, including concrete views, are ad-hoc. The event causing the production of new results for continuous queries is the arrival of a new tuple in the data stream, while for ad hoc queries it is the arrival of the query itself. Thus, joins of a data stream with relations can also be modelled as sequence of ad hoc selection queries—one query for one incoming data stream tuple.

3.1 Window Joins on Data Streams

Say that we are interested in finding buyers who, in two hours, have raised their offers on a given item by more than 20%. Then we could join the **OKBids** stream with a concrete view like that of Example 12. Alternatively, the user can express this query by the ESL statement of Example 14, below, which is more concise and more conducive to efficient implementation.

Example 14 *Buyers who have Raised their Offers by more than 20% in Two Hours.*

```
CREATE STREAM higher_price AS
```

```
SELECT ItemID, SellerID, Price
FROM  OKBids AS H,
      TABLE(OKBids OVER
            (RANGE 2 HOURS PRECEDING CURRENT)) AS L
WHERE H.itemID = L.itemID AND H.bidderID = L.bidderID
AND H.Price  > 1.2 * L.Price
```

Thus, TABLE(**OKBids** OVER ...) generates a TABLE-like view of the stream **OKBids** over logical window defined using the RANGE statement; alternatively we could have used a physical window with the ROWS clause.

The next query illustrates how the window in one stream can be synchronized with the timestamps in another stream. Say that we are interested in finding auctions that closed within 24 hours of opening. Then, we can define a table function that produces a concrete view of **OpenAuction** data stream over a 24-hour window and join the **Closed Auction** data stream with this concrete view:

Example 15 Short Auction Query: *Report all auctions which closed within 24 hours of their opening.*

```
SELECT CA.itemID, CA.current_time
FROM ClosedAuction AS CA,
     TABLE(OpenAuction OVER
          (RANGE 24 HOURS PRECEDING CA)) AS O
WHERE O.itemID = CA.itemID;
```

Thus for each arriving **ClosedAuction** tuple, with timestamp t, we now collect in the window the **OpenAuction** tuples that have arrived in the last 24 hours before t. A more relaxed, and less expensive synchronization will instead be achieved if we change PRECEDING **CA** to PRECEDING **CURRENT**.

Data Stream Joins

At the present time, ESL does provide special constructs to support the symmetric window join of two data streams A and B — i.e., the situation where the arriving tuples in A are joined with those in a window on B, and vice versa. This symmetric join operation can be easily expressed as A joined with a window on B UNION B joined with a window on A. For instance, the query in Example 15 can be transformed into a symmetric join by also joining **OpenAuction** with a window on **ClosedAuction** and then taking the union of the two. But since the closing of an auction always follows its opening, the second join is likely to be empty and we can omit it. It is also easy to see that in the case of self-joins the asymmetric join such as that of Example 14 always produces the same results as the symmetric one. Furthermore, in the next section we will show that in many practical situations it is far more efficient to express join-like operations on streams via UDAs, rather than symmetric joins. Because symmetric joins are not needed in many examples, and our minimalist's bias, they are not supported in the current version of ESL (but they will be added in later versions if the demand for them proves strong).

4 Base Aggregates: Blocking vs. Non-Blocking

In the previous sections, we have concentrated on providing unified support for continuous queries and ad hoc queries, while minimizing extension to SQL:2003. We next focus on extending the query power of ESL to support efficiently a larger class of data stream applications. We show that this can be achieved by UDAs that:

- can express both blocking and non-blocking query operators,
- are powerful and flexible, and can be written natively in ESL itself or in external programming languages,
- are amenable to very efficient implementations on both logical and physical windows using the ESL constructs introduced in the next section.

ESL implements SQL:3's idea of creating a new UDA by specifying the computation to be performed in the three different states called INITIALIZE, ITERATE, and TERMINATE. In ESL, these computations can be specified in an external PL, but can also be defined in ESL itself. In our examples, we will use UDAs of these second type, since they represent a unique feature of ESL and they produce code that is clear and concise.

For instance, Example 16 defines a UDA equivalent to the standard AVG aggregate in SQL. The second line in Example 16 declares a local table, **state**, where the sum and count of the values processed so far are kept. Furthermore, while in this particular example **state** contains only one tuple, it is in fact a table that can be queried and updated using SQL statements and can contain any number of tuples. These SQL statements are grouped into the three blocks labeled, respectively, INITIALIZE, ITERATE, and TERMINATE. Thus, INITIALIZE inserts the value taken from the input stream and sets the count to 1. The ITERATE statement updates the tuple in **state** 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 the computation by the INSERT INTO RETURN statement[5]. Since the TERMINATE statements are processed just after all the input tuples have been exhausted, the UDA in Example 16 below is blocking. Note that the ESL system has built-in support for standard aggregates such as avg, max, min, sum, etc. Aggregate avg is used here for illustration purposes only.

Example 16 *Defining the Standard Aggregate Average*

```
CREATE AGGREGATE avg(Next Real) : Real
{    TABLE state(tsum Real, 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;
     }
}
```

A continuous, non-blocking version of avg can instead be defined as shown in Example 17, below, where the new values are given a higher weight than the old values to assure exponential decay of their importance.

[5]To conform to SQL syntax, RETURN is treated as a virtual table; however, it is not a stored table and cannot be used in any other role.

Example 17 *Continuous Average with Exponential Decay*

```
CREATE AGGREGATE decay_online_avg(Next Real) : Real
{   TABLE state(tsum Real, cnt Int);
    INITIALIZE : {
       INSERT INTO state VALUES (Next, 1);
       INSERT INTO RETURN VALUES (Next);
    }
    ITERATE: {
       UPDATE state
          SET tsum= 0.9*tsum + 1.1*Next, cnt=cnt+1;
       INSERT INTO RETURN
          SELECT tsum/cnt FROM state
    }
    TERMINATE : { }
}
```

UDAs, such as those of Example 17 where the TERMINATE state is empty or missing all together, are non-blocking and can be applied freely to data streams. ESL also uses a (non-blocking) hash-based implementation for the GROUP-BY calls of the UDAs as opposed to the common implementation for SQL aggregates, which first sorts the data according to the GROUP-BY attributes and thus is a blocking operation. This default operational semantics of ESL leads to a stream-oriented execution, whereby the input stream is pipelined through the operations specified in the INITIALIZE and ITERATE clauses.

The UDAs defined so far are called *base* aggregates and follow the syntactic rules of the traditional SQL:2 aggregates. Therefore, they can be used without a group-by clause as in Example 5, or with a group-by clause as shown in Example 18 below.

Example 18 *Find the Recent Average Price of the Items Offered by Each Seller.*

```
CREATE STREAM AskdPrice AS
    SELECT sellerID, decay_online_avg(start_price) AS avg_price
    FROM OpenAuction
    GROUP BY sellerID
```

From a theoretical standpoint, it was known that UDAs make SQL Turing complete on databases, insofar as they can express every function on the database computable by a Turing Machine [20]. Similarly, non-blocking UDAs make SQL complete for data stream applications insofar as it can express every non-blocking query expressible in any other possible language [20]. Our practical experience with ESL indicates that UDAs can be used to implement efficiently mining functions, sequence queries and optimal graph algorithms that cannot be expressed well in SQL:2003. In the next example, we show how a non-blocking UDA can be used to save memory in the computation of self-joins.

Self-joins

Since data streams are unbounded, naive joins over streams might require infinite memory. Say that **calls(call_ID, event, time)** is a stream of phone-call records, where we use **start** or **end** in the **event** field to indicate whether **time** field marks the beginning or the end of a phone-call. Then to find the length of every conversation, we could self-join the stream with itself to find two tuples that have the same **call_ID**, and their **event** values are respectively **start** and **end**.

Example 19 *List the Length of Every Call in* calls(call_ID, event, time)

```
        SELECT O1.call_ID, O2.time - O1.time
        FROM calls AS O1, calls AS O2
        WHERE ( O1.call_ID = O2.call_ID
                AND O1.event = 'start'
                AND O2.event = 'end');
```

However, this statement could require infinite memory. One approach to solve this problem is to join the data stream **O1** with a window on the data stream **O2**, as done in Examples 14 and 15. But this would only provide an approximate solution, since the exact length of a conversation whose duration is longer than the window size cannot be reported. A better solution consists in deleting from the window those records whose 'end' has already been seen. This allows us to compute the duration with bounded memory, given that the number of calls drop exponentially with increasing call length, as in the following UDA:

Example 20 *List the Length of Every Call in* calls(call_ID, event, time)

```
CREATE AGGREGATE call_len(callid, event, time) : (Id, Length)
{   TABLE memo(id, start);
    INITIALIZE: ITERATE: {
        INSERT INTO memo VALUES(callid,time)
        WHERE Event='start';
        INSERT INTO RETURN SELECT id, time - start
           FROM memo WHERE event='end'
            AND memo.id=callid;
        DELETE FROM memo
           WHERE event='end' AND memo.id=callid;
    }
}
```

The UDAs discussed in this section are base UDAs: base UDAs with an empty or missing TERMINATE clause are non-blocking and can be applied freely to on data streams; blocking UDAs, instead, can be used freely on database relations, but they can be applied on data streams only through the window construct, discussed in the next section. Window aggregates enhance the performance of UDAs and the user convenience, thus they are part of ESL, although their functionality could be captured directly using base UDAs.

For instance, Example 21 below shows the definition of an aggregate that behaves as the **unlimited preceding** version of average.

Example 21 *The Cumulative Average–i.e., its 'unlimited preceding' version*

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

This cumulative version of **avg** was obtained from its base definition in Example 16, by taking the return clause from TERMINATE and appending it to INITIALIZE and ITERATE. The UDA so obtained has an empty TERMINATE clause and it is thus non-blocking and also efficient (at least to the extent in which the base UDA is). Therefore, the ESL compiler uses this rewriting technique to implement 'unlimited preceding' versions of UDAs [6].

Nevertheless, for general windows on UDAs the implementations that can be derived from their base definition tend to be inefficient and require a different solution for this problem, which is discussed in the next section.

5 Window Aggregates

ESL supports the optimization of window aggregates both at the physical and logical level by:

- the **inwindow** construct whereby the system optimizes the management of windows, and
- the CREATE WINDOW AGGREGATE declarations, whereby the user can specify an optimized implementation for each window aggregate via the EXPIRE construct.

The use of the CREATE WINDOW AGGREGATE declaration and the **inwindow** construct is illustrated by Example 22 below, which defines a naive implementation of **avg** over a finite-sized window.

Example 22 *A Naive Version of* **avg** *on a Finite Window*

```
CREATE WINDOW AGGREGATE avg(Next Real) : Real
{
  TABLE inwindow(wnext Real);
  INITIALIZE :
  ITERATE : {
    INSERT INTO RETURN
      SELECT avg(wnext) FROM inwindow;
      }
}
```

Observe that, in Example 22, our window version of AVG calls on the base **avg** aggregate for tables; but this is not a recursive call, since the two aggregates are internally treated as two different procedures.

The declaration TABLE **inwindow(wnext** real) instructs the system to store the input values in a special window buffer that will be called **inwindow**[7]. Incoming tuples (expiring tuples) are automatically added to (deleted from) **inwindow** by the system. The system performs the window maintenance task on behalf of the user, with the same interface to both physical and logical windows. This unification makes it easier to specify window UDAs. Moreover, it creates opportunities for physical optimization by sharing windows between UDAs and swapping large windows to secondary store when necessary[22]. Although the use of **inwindow** assures efficient storage management, the algorithm shown in Example 22 is still inefficient since it recomputes **avg** on the current window for each new incoming tuple. It takes time $O(K \times n)$, where n is the number of tuples in the input and K is the number of tuples in **inwindow**.

To further optimize such window aggregates ESL makes available to users the EXPIRE construct, discussed next, which enables computation in time $O(n)$. Example 23 defines a highly optimized implementation of AVG, using a new state called EXPIRE in which the values of tuples leaving the window are used to perform delta-maintenance on the window UDA. For AVG, the delta maintenance consists in decreasing the sum by the value of the expired tuple and the count by 1. The result is the same whether this delta computation is performed as soon as a tuple expires, or later when a new tuple comes in, or anywhere in between these two instants. ESL takes advantage of this freedom to optimize execution.

Example 23 *The New Construct* EXPIRE

```
CREATE WINDOW AGGREGATE myavg(Next Real) : Real
{ TABLE state(tsum Int, cnt Real);
  TABLE inwindow(wnext Real);
  INITIALIZE : {
    INSERT INTO state VALUES (Next, 1);
    INSERT INTO RETURN VALUES (Next);
  }
  ITERATE : {
    UPDATE state SET tsum=tsum+Next, cnt=cnt+1;
    INSERT INTO RETURN
      SELECT tsum/cnt FROM state;
  }
  EXPIRE: {
    /* when there are expired tuples take the oldest */
    UPDATE state SET cnt= cnt-1,
           tsum = tsum - oldest().wnext
  }
}
```

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. In ESL, **oldest()** is a built-in function that delivers the oldest among the tuples in **inwindow**, and **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 coulmn name.

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

The following examples show how specialized versions of window aggregates can result in significant performance improvements, and ESL users are likely to take full advantage of this option in their applications. However, users are not required to define the window version of a UDA, since ESL falls back on the base version whenever the window version is not provided. For instance, in the absence of definition Example 23, the ESL system uses the definition of Example 22 to support the application of **avg** over finite windows, and for 'unlimited preceding' windows ESL instead uses the definition in Example 21. This policy is applied uniformly to all UDAs, not just to **avg** and it is made possible by the fact that once a base definition exists for any UDA, the 'unlimited preceding' and the 'finite window' versions can be trivially derived by simple syntactic

[6]In fact, instead of rewriting the UDA, ESL simply executes the TERMINATE statement after each INITIALIZE or ITERATE execution.

[7]The names of the columns of **inwindow** can be chosen by the user, but their data types must be the same as the aggregate arguments.

rewriting. This convenience also holds when PL statements are used to define the computations performed in the three states INITIALIZE, ITERATE, TERMINATE (Access to **inwindow** by PL functions is provided by our system for this purpose).

The table **inwindow** is managed by the system, which inserts arriving tuples and deletes expiring tuples according to the window range and its type (i.e., logical or physical). Insertion of new tuples in the window by ESL statements is not allowed since it is incompatible with the window semantics that assume externally arriving tuples rather than internal ones. However, there is no reason that tuples that are no longer needed must be kept in the window until they expire out of the window range: therefore besides unrestricted queries on **inwindow**, ESL also allows the deletion of inwindow tuples as part of the UDA statements. The application of this is illustrated by the **sumdistinct** aggregate: when a new tuple arrives we can eliminate older duplicate values from the window.

Example 24 *Sum Distinct with Windows*

```
CREATE WINDOW AGGREGATE sumdistinct (Next Real) : Real
{    TABLE thesum(tsum real);
     TABLE inwindow(wnext real);
     INITIALIZE : {
        INSERT INTO thesum VALUES (Next);
        INSERT INTO RETURN VALUES (Next);
     }
     ITERATE : {
        DELETE FROM inwindow WHERE wnext=Next;
        UPDATE thesum SET tsum=tsum+Next
        WHERE SQLCODE <> 0;
        INSERT INTO RETURN
           SELECT tsum FROM thesum;
     }
     EXPIRE: {
        /* when there are expired tuples take the oldest */
        UPDATE thesum SET tsum = tsum - oldest();
     }
}
```

Consider now the window version of **max**: for each incoming tuple, we can eliminate the older tuples of less or equal value. Thus the oldest tuples in the window are also those that have the max value which is therefore returned as the result of the aggregate.

Example 25 *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*/
}
```

The optimized treatment of window aggregates find many natural applications. Let us return to the out-of-order problem of Example 10, where we have shown how tuples that are less than two minutes late can be reordered by calling **resort_bids** with a window of two minutes. The windowed version of **resort_bids** is defined in Example 26: whenever a tuple expires, we return the tuple along with any tuples that have **bid_time** less than or equal to the expiring tuple's **bid_time**. This UDA does not reorder tuples that are more than 2 minutes out-of-order, which will be redirected to the out-of-order stream as discussed in Section 2.3. Alternatively, it is easy to modify the UDA of Example 26 to either delete these tuples or to reassign their timestamps to the max value seen so far.

Example 26 *UDA to Resort Late Arrivals in a Window*

```
CREATE WINDOW AGGREGATE resort_bids (itemID Int,
     bid_price Real, bidderID Int, bid_time Timestamp):
         (itemID Int, bid_price Real, bidderID Int,
         bid_time Timestamp)
{    TABLE inwindow(itemIDw Int, bid_pricew Real,
                    bidderIDw Int, bid_timew Timestamp);
     INITIALIZE : { } /* the system adds new tuples to inwindow */
     ITERATE : { }
     EXPIRE: {
        INSERT INTO RETURN
           SELECT itemIDw, bid_pricew, bidderIDw, bid_timew
           FROM inwindow
           WHERE bid_timew ≤ oldest().bid_timew
           ORDER BY bid_timew;
        DELETE FROM inwindow WHERE bid_timew ≤ oldest().bid_timew;
     }
}
```

The above UDA first collects the tuples in the **inwindow** buffer, and no other processing is done in the INITIALIZE and ITERATE states. Then, when a tuple expires, all tuples that are younger than the expiring tuple are returned, i.e. in the EXPIRE state. Reordering mechanism defined by Examples 26 and 10 together is very similar to the slack mechanism in the Aurora system [4], however ESL allows more flexibility to the user by allowing them to handle out-of-order tuples in any way they like, not just reorder or discard them. These examples illustrate the desirability of customizing delta-maintenance on each UDA. Remarkably, the declarative framework here proposed is effective on both physical and logical windows, and, in addition to data streams, it can also be applied to database tables.

Data stream mining applications discussed next illustrate another interesting application of window aggregates. In these examples, the performance obtained by coding UDAs in ESL was comparable to that obtained by coding the UDAs in C++ (typically, a 10% slow-down).

6 Applications

In [20] we proved that (i) the introduction of natively defined UDAs turn SQL into a Turing-complete language, and (ii) the language is also complete for data stream applications since it can express all the monotonic queries expressible by a Turing machine (non-monotonic queries cannot be used on data streams since they are always blocking). The theoretical results presented in [20] are important, but, in the end, the strength of a language and its system can only be evaluated through actual applications. ESL and the Stream Mill system [2] have proven effective on a wide range of complex applications that includes (i) Mining Data Streams [21], (ii) Sequence Queries [10], (iii) Streaming XML queries [32], and (iv) Approximate Computations.

Mining Data Streams

Support for data stream mining represents a major problem for DSMS. The problem has its roots in the inadequacy of SQL-compliant DBMS to support data mining methods, which have been the focus of much previous research. This problem was solved in Wang et al. [29], where it was shown that UDAs are effective at expressing data mining methods, by expressing the complex statistical computation that are at the core of these methods. In [21], we show that ESL UDAs are particularly effective on data streams, where mining methods tend to perform one pass over the data (rather than many passes), have to adapt continuously to concept drifts, and make use of windows and other synopses.

Sequence Queries

Several query languages have been proposed in the past for sequences and time-series [23, 26]. The main motivation of these languages is that most self-join queries are too complex to express and inefficient to implement if they are written in SQL [23, 26]. Take for instance the following query on time-series of daily temperatures: Find a fortnight of raising temperatures. In SQL, this query requires 14 joins; that many joins cannot be supported efficiently, and are also too complex for users to write. Other queries, such as double-bottom queries [26] would require an unbounded number of self-joins, and might not be expressible in SQL. All these queries can however be expressed in ESL using UDAs [10].

In the following example we have a log of web pages clicked by a user, as follows:

```
CREATE STREAM
    Sessions(SessNo Int, ClickTime Timestamp, PageNo Int,
        PageType Char);
    ORDER BY ClickTime
```

A user entering the home page of a given site starts a new session that consists of a sequence of pages clicked. For each session number, **SessNo**, the log shows the sequence of pages visited—where a page visit is described by its timestamp, **ClickTime**, number, **PageNo** and type, **PageType** (e.g., an advertisement page, a product description page, or a page used to purchase the item). The ideal scenario for advertisers is when users (i) see the advertisement page for some item in a content page, (ii) jump to the product-description page with details on the item and its price, and finally (iii) click the ‘buy this item’ page. This advertisers’ dream pattern can be expressed by the following UDA query, where ‘ad’, ‘pd’, and ‘buy’, respectively, denote an ad page, a product description page, and a purchase page. We store the last three page types using the memo table in the UDA **find_pattern**. Then we issue a self-join query to check the condition. Since the memo table only contains three tuples at all times, the self-join query is very efficient.

Example 27 *Sequence Query*

```
SELECT SessNo, find_pattern(PageType)
FROM Sessions
GROUP BY SessNo;
```

```
AGGREGATE find_pattern(PageType Char):
{   TABLE memo(PageType Char, state Int) MEMORY;
    INITIALIZE: ITERATE: {
        UPDATE memo SET state = state + 1;
        DELETE FROM memo WHERE state = 3;
        INSERT INTO memo VALUES (PageType, 0);
        INSERT INTO RETURN
            SELECT 'Found Pattern'
            FROM memo AS X, memo AS Y, memo AS Z
            WHERE X.state = 0 AND X.PageType = 'buy'
                AND Y.state = 1 AND Y.PageType = 'pd'
                AND Z.state = 2 AND Z.PageType = 'ad';
    }
}
```

Support for Continuous Queries on Streaming XML

The need to unify the support of relational streams and XML streams into one DSMS is very strong, since it is driven by application demands similar to those that are now pushing DBMS vendors to support stored XML documents. Indeed, many competing solutions have been proposed for the unified management of DB relations and XML. However, at the best of our knowledge, Stream Mill is the only DSMS that currently supports both kinds of streams. Indeed, the expressive power of ESL greatly simplified the difficult task of supporting XML streams and XQueries in our system [32].

Approximate Computation on Data Streams

A new requirement introduced by data streams is the need to support approximate computation and synopses. Many of these can be naturally expressed as UDAs. For instance Datar et al [14] proposed an elegant solution to the following problem:

Given a stream of data elements consisting of 0’s and 1’s, maintain an approximate count of the number of 1’s in the last N elements using as little memory as possible.

The solution proposed by Datar et al [14] creates a new bucket of value 1, for each incoming element of value 1. If there are k neighboring buckets with same value v, we merge the oldest two buckets into one bucket with value $2v$ (value k depends on the error bound ϵ, which is given by the user). At any time, the approximate count is the sum of the values of all buckets minus half of the value in the oldest bucket. It can be shown that the algorithm maintains an approximate count with a relative error bound of $1+\epsilon$ with $\Omega(\frac{1}{\epsilon}\log^2 N)$ bits of memory space [14].

The above algorithm is realized easily by UDA BasicCount in ESL:

Example 28 *Maintaining Approximate Counts over Streams*

```
TABLE hist(h Int, timestamp Int) AS VALUES (0, 0);
CREATE AGGREGATE BasicCount(v Int, timestamp Int) : Int
{   INITIALIZE : ITERATE : {
        INSERT INTO hist VALUES (v, timestamp);
        DELETE FROM hist AS h
            WHERE h.timestamp < timestamp - N;
        INSERT INTO RETURN
            SELECT merge(h, timestamp)
            OVER (ORDER BY timestamp DESC) FROM hist;
    }
}
```

In Example 28, we assume elements arrive with increasing timestamps $1, 2, 3, \cdots$, and we use a table, hist, to maintain the buckets. Initially, the histogram contains a dummy entry $(0, 0)$. We add new entries to table hist for elements of value 1, and we remove expired entries from the window. We merge histograms by invoking UDA merge in Example 29, which iterates through entries in hist, starting from the last entry to the dummy entry $(0, 0)$.

Example 29 *Merge Duplicate Values in the Histogram*

```
CREATE AGGREGATE merge(iValue Int, iTime Int):
{    TABLE state(v Int, c Int, t Int) AS VALUES(1,0,0);
     INITIALIZE: ITERATE : {
        UPDATE state SET c=c+1,t=iTime
           WHERE v=iValue;
        UPDATE state SET v=iValue,c=1,t=iTime
           WHERE v<>iValue AND c≥k;
        UPDATE hist SET h = h * 2 WHERE SQLCODE=0
           AND timestamp=(SELECT t FROM state);
        DELETE FROM hist WHERE SQLCODE=0
           AND timestamp=(SELECT t-1 FROM state);
     TERMINATE : {
        INSERT INTO RETURN
           SELECT sum(h)-iValue/2 FROM hist;
     }
}
```

In UDA merge, we count if a value has more than k entries in the histogram. If it does, we double the value of the current entry, delete the last entry, and restart counting. Finally, we return the approximate count.

7 Related Work and Discussion

There are obvious similarities between the ESL approach and that of other data stream projects, but also significant differences. A first difference is that ESL strives to unify the treatment of continuous queries and ad hoc queries by building as much as possible on standard syntax and semantics. For instance, SQL:2003 supports windows as aggregate modifiers in OLAP Functions, providing constructs that can be applied directly to continuous queries on data streams. Moreover, ad hoc queries on both database tables and tables created as windows on data streams can be supported using SQL:2003 constructs of concrete views and table functions. These constructs can thus be used in ESL to specify window joins on streams.

While conservatively conforming to SQL:2003 syntax, ESL has boldly attacked its expressive power problems, and has extended it into a language that can be used to support complex queries such as time series queries. The basic idea was taken from [20] where it was shown that UDAs can turn SQL into a Turing complete (and non-blocking complete) query language. ESL has materialized the abstract notions of UDAs from [20] into a flexible and efficient tool for data stream processing, by providing delta-based maintenance mechanisms whereby arbitrary UDAs can work efficiently on windows of different kinds. The topic of optimizing windows was the subject of interesting previous work, which primarily focused on the window sharing problem [7, 30, 15].

The issue of flexible timestamp management has been studied in [27], where internal and external timestamps were considered, and sophisticated techniques to handle out-of-order tuples were proposed. The question on whether these techniques can be implemented in ESL, by using system generated heartbeats and application-specific UDAs represents an interesting topic for further research. Also, punctuation [24], that has proven very useful in dealing with blocking aggregates, can be used in a role similar to heartbeats, particularly for data streams with latent timestamps.

Instead of extending SQL, Aurora defines query plans via an attractive "boxes and arrows" graphical interfaces [12]. Aurora supports eight primitive operations, among which four are windowed operators. It also supports user-defined aggregates, which are defined in a procedural language. The aggregation operators have optional parameters, making their semantics dependent on stream arrival and processing rates.

GSQL is a pure stream query language designed for Gigascope, a stream database for network applications [13]. GSQL is an SQL-like language allowing query composition and query optimization, which supports sort-merge union of streams, joins of two streams, and aggregation, besides externally defined functions.

TelegraphCQ [15] proposes a SQL-like language with extended window constructs, including a low-level C/C++-like for-loop, aiming at supporting more general windows such as backward windows. TelegraphCQ [15], and also CQL [6], treat windows as stream modifiers, rather than aggregate modifiers as SQL:2003 does.

The main restriction of ESL with respect to other languages is that it does not contain special constructs to express window joins on multiple streams directly. This did not prove to be a major problem in the many complex applications we have implemented so far. However, if needed, the various kinds of window joins proposed in the literature [19, 28, 18, 5] can be easily added since table expressions in SQL:2003 already accommodates special joins, such as outer-joins.

8 Conclusion

Continuous query languages that comply with SQL:2003 standards are preferable for database vendors and database researchers alike, and will simplify applications that span both DB tables and data streams. However, continuous queries on streaming data are so different from traditional applications on stored data that other DSMS projects have much deviated from the standards [6], or replaced SQL all together [4]. The design of ESL shows that

- SQL:2003 syntax and semantics can be used to unify continuous queries and ad hoc queries in a quite natural fashion (using concepts such as concrete views and table functions)
- UDAs enhanced with window constructs are necessary and sufficient to support a wide range of advanced data stream applications,
- UDAs can be defined natively and concisely in SQL itself.

The conservative approach taken by ESL, with respect to standards, has produced a language of great power and generality, which supports efficiently (i) continuous and ad-hoc queries on data streams, (ii) data stream mining, (iii) sequence queries, (iii) queries on streaming XML, and (iv) approximate computations.

At the time of this writing, the ESL system is fully operational and supports continuous and ad hoc queries using a client-server architecture [2].

References

[1] The IBM Websphere software. http://www-306.ibm.com/software/websphere.

[2] Stream mill home. http://wis.cs.ucla.edu/stream-mill.

[3] The Teradata database. http://www.teradata.com.

[4] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2):120–139, 2003.

[5] M. Riedewald: Abhinandan Das, J. Gehrke. Approximate join processing over data streams. In *SIGMOD*, pages 40–512, 2003.

[6] A. Arasu, S. Babu, and J. Widom. Cql: A language for continuous queries over streams and relations. In *DBPL*, pages 1–19, 2003.

[7] Arvind Arasu and Jennifer Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, pages 336–347, 2004.

[8] Jennifer Widom Arvind Arasu. A denotational semantics for continuous queries over streams and relations. *SIGMOD Record*, 33(3):6–12, 2004.

[9] B. Babcock, S. Babu, M. Datar, R. Motawani, and J. Widom. Models and issues in data stream systems. In *PODS*, 2002.

[10] Yijian Bai, Chang Luo, Hetal Thakkar, and Carlo Zaniolo. Efficient support for time series queries in data stream management systems. http://wis.cs.ucla.edu/publications/eslTS.pdf, 2004.

[11] D. Barbara. The characterization of continuous queries. *Intl. Journal of Cooperative Information Systems*, 8(4):295–323, 1999.

[12] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB*, Hong Kong, China, 2002.

[13] C. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: High performance network monitoring with an sql interface. In *SIGMOD*, page 623. ACM Press, 2002.

[14] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows: (extended abstract). In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 635–644, 2002.

[15] Sirish Chandrasekaran et al. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[16] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *ACM SIGMOD Record*, 32(2):5–14, 2003.

[17] Eric N. Hanson. A performance analysis of view materialization strategies. In *SIGMOD*, page 440, 1987.

[18] Jaewoo Kang, Jeffrey F. Naughton, and Stratis Viglas. Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352, 2003.

[19] M. T. Ozsu: L. Golab. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, pages 500–511, 2003.

[20] Yan-Nei Law, Haixun Wang, and Carlo Zaniolo. Data models and query language for data streams. In *VLDB*, pages 492–503, 2004.

[21] C. Luo, H. Thakkar, H. Wang, and C. Zaniolo. A native extension of sql for mining data streams. ACM SIGMOD 2005 demo paper.

[22] Chang Luo, Haixun Wang, and Carlo Zaniolo. Efficient support for window aggregates on data streams. http://wis.cs.ucla.edu/publications/windows.pdf, 2004.

[23] Raghu Ramakrishnan Praveen Seshadri, Miron Livny. The design and implementation of a sequence database system. In *VLDB*, pages 99–110, 1996.

[24] P.Tucker, D. Maier, and T.Sheard. Applying punctuation schemes to queries over continuous data streams. *IEEE Data Engineering Bulletin*, 26(1):33–40, 2003.

[25] Berthold Reinwald, Hamid Pirahesh, Ganapathy Krishnamoorthy, George Lapis, Brian Tran, and Swati Vora. Heterogeneous query processing through sql table functions. In *ICDE*, pages 366–373, 1999.

[26] Reza Sadri, Carlo Zaniolo, Amir Zarkesh, and Jafar Adibi. Optimization of sequence queries in database systems. In *PODS*, Santa Barbara, CA, May 2001.

[27] Utkarsh Srivastava and J. Widom. Flexible time management in data stream systems. In *PODS*, pages 263–274, 2004.

[28] M. J. Franklin Tolga Urhan. Xjoin:a reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, 2000.

[29] Haixun Wang and Carlo Zaniolo. Atlas: a native extension of sql for data minining. In *Proceedings of Third SIAM Int. Conference on Data Mining*, pages 130–141, 2003.

[30] Haixun Wang, Carlo Zaniolo, and Chang R. Luo. ATLaS: A small but complete sql extension for data mining and data streams. In *VLDB*, pages 1113–1116, 2003.

[31] Fred Zemke, Krishna Kulkarni, Andy Witkowski, and Bob Lyle. Proposal for OLAP functions. In *ISO/IEC JTC1/SC32 WG3:YGJ-nnn, ANSI NCITS H2-99-155*, 1999.

[32] Xin Zhou, Hetal Thakkar, and Carlo Zaniolo. Unifying the processing of xml streams and relational data streams. Concurrent submission to this conference, 2005.