An Introduction to the Expressive Stream Language (ESL) [1]

WEB Information System Laboratory
UCLA, CS Department
`http:\\wis.cs.ucla.edu`

Abstract

ESL is the application language of the Stream Mill system that supports[2]:

- Continuous queries on data streams,
- Ad hoc queries on (i) database tables and (ii) on concrete views created from streaming data,
- Spanning applications that combine and compare incoming live data with stored data.

ESL is based on SQL to help users to learn it, and use it on spanning applications. For the same reasons, ESL strives to minimize the SQL extensions introduced to handle data streams (rather than database tables). Within this minimalist's approach, however, ESL provides a quantum leap in expressive power: ESL is Turing complete and can support efficiently applications that are well beyond the capabilities of SQL:2003 —e.g., Data Mining queries and time series queries.

This report is as short tutorial on ESL for simple data stream applications; the reader should refer to `http:\\wis.cs.ucla.edu` for more advanced applications, including data mining queries and time series queries.

[1]The main contributors to the design and implementation of ESL and Stream Mill are: Yijian Bai (bai@cs.ucla.edu), Richard Chang Luo (luo@cs.ucla.edu), Hetal Thakkar (hthakkar@cs.ucla.edu), Haixun Wang (haixun@us.ibm.com), and Carlo Zaniolo (zaniolo@cs.ucla.edu)

[2]Feature not yet implemented.

Contents

1 Streams and Timestamps

ESL views data streams as unbounded ordered sequences of tuples [4]; this is consistent with the 'append only table' model commonly used by data stream systems. In Stream Mill, each data stream is imported from an external wrapper via the (mandatory) SOURCE clause in its CREATE STREAM declaration. This declaration also specifies kind of timestamp associated with the stream. ESL supports the following three kinds of timestamps: (i) *external timestamps*, (ii) *internal timestamps*, and (iii) *latent timestamps*.

External timestamps are values that are contained in the arriving tuples (e.g., placed there by application producing the data); therefore all 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** declaration in Example 1, below, is instead assigned an internal timestamp, generated by recording the current time in a new column added to the incoming tuple. This new column is always 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.

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 /*this creates internal timestamps*/)
    ORDER BY current_time;  /* internal timestamp */
    SOURCE 'POR4446';
CREATE STREAM Bid( /* Bid: Stream of bidding.*/
    itemID int /* the item being bid for*/,
    bid_price real, /* bid price */,
    bidderID char(10) /* id of the bidder*/,
    bid_time timestamp /* time when bid was registered */)
    SOURCE 'port4447';
```

Finally, *latent timestamps* are simply declared by omitting the ORDER BY clause in the CREATE STREAM declaration. While the values of internal timestamps are made explicit eagerly as tuple enter the system, the values of latent timestamps are instantiated lazily and only when they are needed in operations whose semantics depends on explicit timestamps (e.g., union or window aggregates). For most operations, only the order of the tuples is of importance, and the order can be supported without explicit timestamp columns in the actual tuples. Therefore, latent timestamps can reduce the size of the tuples, and deliver the convenience of operational semantics when this is needed.

In our Example 1, above, the data stream **Bid** has latent timestamps —but this is not the only data stream with latent timestamps created by the declarations in Example 1. In fact, a stream with latent timestamps is implicitly created for each externally timestamped stream. In fact, for externally timestamped streams, there is no assurance that their tuples arrive sorted according to their timestamps. ESL solves this problem by reassigning late tuples to a separate stream with latent timestamps, which can then be handled directly by the user. The name of these streams is that of the externally timestamped streams with the postfix **outOforder**. Therefore, the first declaration in Example 1 also produces the data stream **OpenAuction_outOforder**, which has the same attributes as **OpenAuction** but latent timestamps.

Example 1 above 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 Unix end-of-line characters: `\n`. Rather than using these defaults, users can easily create their own wrappers as described in [2].

2 Single Stream Transducer and Queries

In ESL, new streams can be defined from existing streams in a fashion similar to that used to define virtual views in SQL. For instance, to derive 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
```

Since the explicit timestamp column **start_time** is preserved, the resulting stream is viewed as having explicit timestamps—but if **start_time** were projected out, then **expensiveItems** would be viewed as having latent timestamps.

New internal streams can now be defined from **expensiveItems** as per the usual cascading of virtual views upon virtual views. An INSERT INTO statement is instead used to specify that tuples must be appended into the output stream STDOUT (or other explicitly declared streams.) For instance, the query of Example 3, below, inserts results into the output stream to be returned to the user[3]:

Example 3 *Sending the results of a continuous query to the output*

```
INSERT INTO STDOUT
SELECT itemID, start_price, start_time
FROM expensiveItems WHERE sellerID= 'Jane Doe'
```

Joins with DB Tables. ESL also supports joining data streams with database relations. For instance, if we have the database table

sellerinfo(sellerID, ZipCode, ...)

then the following query can be used to add the zipcode of the seller to the **expensive_items** streams:

Example 4 *Joining a data stream with a database table*

```
CREATE STREAM offersZC
SELECT ZipCode, itemID, start_price, start_time
FROM expensive_items AS I, sellerinfo AS S
WHERE I.sellerID= S.SellerID
```

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 [4]. ESL supports two kinds of aggregates: base aggregates, and window aggregates.

Example 5 uses of a base aggregate called **decay_online_avg** to compute the exponential decay of the closing values of auctions. As discussed in Section 5, this aggregate is a nonblocking aggregate, since blocking aggregates cannot be applied directly to data streams.

For blocking aggregates (and the traditional standard SQL-2 aggregates are blocking), only their window versions can be used on data streams. While base aggregates are called as the traditional SQL-2 aggregates (with possibly a group-by clause), window aggregates are called with the OVER clause of SQL:2003 OLAP functions. For instance, Example 6, below, shows the use of an unlimited window, whereby the **max** returns the highest timestamp seen so far.

[3]Here, the INSERT INTO construct really means 'append to the end' (of the data stream). ESL does no allow DELETE and UPDATE statements on data streams.

Example 5 *The Recent average of the closing bids*

```
    CREATE STREAM recentaverage
    SELECT decay_online_avg(final_price)
    FROM ClosedAuction
```

Example 6 *A solution for the out-of-order problem*

```
    CREATE STREAM bidsInorder
    SELECT itemID, bid_price, bidderID, max(bid_time)
            OVER(RANGE UNLIMITED PRECEDING) AS NewTime
    FROM Bid
```

Thus Example 6 provides a simple but effective solution to the out-of-order timestamp problem for our **bid** data stream. where tuples arriving late are assigned the highest timestamp value seen so far.

3 UNION

Union is the only ESL operator that is directly applicable to multiple data streams. For instance, the query in Example 7, below, sort-merges two streams according to their explicit timestamps into a data stream called **history1000**.

Example 7 *Price History Query: Report price history for item #1000*

```
CREATE STREAM history1000 AS
    SELECT itemID, start_price, start_time
    FROM OpenAuction
    WHERE itemID = 1000
    UNION
    SELECT itemID, bid_price, NewTime
    FROM bidsInorder
    WHERE itemID = 1000
```

In general, UNION is only supported on union-compatible data streams and the union of a data stream with a DB table is not allowed in ESL. The union of explicitly timestamped data streams produces an explicitly timestamped data stream, as illustrated by Example 7. Likewise, the union of two or more data streams with latent timestamp produces a data stream with latent timestamps. The mixing of data streams with explicit timestamps and latent timestamps is not allowed.

To union data streams with explicit timestamps, we ensure that the resulting stream is ordered by their common time stamps. This is achieved through a special sort-merge operation on the streams. We choose the tuple with the minimum timestamp from the streams if none of the streams is empty. If some 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.

In implementing the union of streams with latent timestamps, the system goes round-robin its input buffers and takes tuples currently there. Therefore, while the order of the tuples is always preserved for each input stream, the order across different streams depends on implementation and load conditions—although the system is designed to service all input buffers fairly this is only assured on a 'best-effort' basis.

Besides UNION, ESL also support the UNION ALL operator, and the two behave differently on explicitly timestamped data streams: UNION eliminates tuples that are identical in their values and timestamps, and UNION ALL does not.

The treatment of UNION on streams with latent timestamps is left to the implementation (where the extra effort required in dealing with duplicates is likely to be avoided).

4 Ad Hoc Queries on Tables and Concrete Views

ESL also supports traditional queries on database tables. These are called ad-hoc queries, to distinguish them from continuous queries that once issued persists until they are turned off by the users. Ad hoc queries are instead off as soon as they they complete, and they remain off until the are re-issued by the user. The ad hoc queries supported in ESL are those of Atlas, and the user is referred to [1] for detailed documentation.

4.1 Concrete Views

Besides ad hoc queries on database tables, ESL supports ad hoc queries on *concrete views* that are created as windows on streams using a syntax similar using the syntax shown in Example 8, below.

Example 8 Concrete View: *A table containing all the auctions where the price was above 1000 for the last 10 minutes.*

```
CREATE TABLE high_priced AS (SELECT ItemID, SellerID
               FROM OpendAuction OVER
                      (RANGE 10 MINUTE PRECEDING CURRENT)
                 WHERE start_price > 1000) REFRESH IMMEDIATE
```

Therefore, RANGE **10** MINUTE PRECEDING CURRENT, clearly denote that we collect all the tuples preceding and including the current tuple. For uniformity with windows on aggregates (Example 6), ESL also allows CURRENT to be omitted without any effect on the semantics.

As illustrated by this example ESL also supports the REFRESH IMMEDIATE options of SQL:2003 concrete views; this specifies that system should maintain this view up-to-date to respond immediately to the query issued by the user [4]

Therefore in ESL, the persistence of queries follow from the object they are applied to. Queries applied to data streams are continuous; queries applied to tables, including concrete views, are ad hoc. Therefore, the following ad hoc query uses a blocking aggregate and returns a single value for each time it is executed.

Example 9 Concrete View Query: *Count the number of high-priced items place in the last 10 Minutes.*

```
INSERT INTO STDOUT
SELECT count(ItemID)
        FROM high_priced
```

Therefore, 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 can be viewed as a sequence of selection queries in response of the incoming data stream tuples.

4.2 Joins

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 **bidsInorder** stream with a concrete view like that of Example 8 (without the where clause). Alternatively, the user could express this query by the ESL statement of Example 10, below, which is more concise and more conducive to efficient implementation.

Example 10 *Buyers have raised their offers offers by more than 20% in the two hours.*

```
 CREATE STREAM higher_price AS SELECT ItemID, SellerID, Price
        FROM  bidsInorder AS H,
        TABLE(bidsInorder OVER RANGE 2 HOUR PRECEDING CURRENT) AS L
WHERE H.itemID = L.itemID AND H.bidderID = L.bidderID
AND H.Price  > 1.2 * L.Price
```

[4] Although this is not supported at the current time, ESL will also support the REFRESH DEFERRED option. In this second case, the system rebuilds the view when the query is received. The response to the query might be slow: in example Example 8 above, it could take up to 10 minutes to rebuild the view.

Thus, TABLE(**bidsInorder** OVER ...) generates a TABLE-like view of the stream **bidsInorder** over logical windows defined using the RANGE statement; alternatively we could have used a physical window by using the ROWS clause. Also, CURRENT can be omitted without changing the meaning of this statement.

This approach is consistent with SQL:2003, where table functions are used to recast practically any kind of external data source into SQL tables [5]. Also the constructs used here to specify windows are based on those used in ESL for window aggregates, which, in turn, are those of SQL:2003 OLAP functions[5].

In the query of Example 10, above, the incoming stream and the window are kept in sync by the system on a best-effort basis. There is no explicit checking of timestamp values. This is changed in the query of Example 11, below:

Example 11 *Synchronized self join.*

```
 CREATE STREAM higher_price AS SELECT ItemID, SellerID, Price
        FROM  bidsInorder AS H,
        TABLE (OpenAuction OVER(RANGE 2 HOUR PRECEDING H)) AS L
WHERE H.itemID = L.itemID AND H.SellerID = L.SellerID
AND H.Price > 1.2 * L.Price
```

In this example, the window is explicitly synchronized with H, by the 'PRECEDING **H**' clause. This specifies that for each new tuple of **H**, all the tuples with a timestamp value that is less or equal to that of **H** must be included in the window being joined. Clearly, this constraint is effectively enforced only when the timestamps of both data streams are explicit.

This example illustrates how self-joins on data stream can be expressed in ESL. In the next example, we consider joins involving two different data streams. Say for instance that we are interested in finding auctions that closed within 24 hours. Then, we can define a table function that produces a concrete view of this data stream over a 24 hour window; then we join the **Closed Auction** data stream with this concrete view:

Example 12 Short Auction Query: *Report all auctions which closed within 24 hours of their opening.*

```
INSERT INTO stdout  SELECT IdemID, curren_time
FROM ClosedAuction AS C,
        TABLE(OpenAuction OVER (RANGE 24 HOUR PRECEDING C)) AS O
WHERE O.itemID = C.itemID;
```

[5]However, the partition-by clause that is supported for aggregates is not supported for table functions.

These examples illustrate how to compute self-joins on a window, and joins for streams such as ClosedAuction that must be matched with a window of previous events. These kind of joins are very common in many applications. The most general kind of join that is possible for two data streams, A and B, is when we match the tuples in A with the previous tuples held on a window on B, and also vice versa. This case can simply be expressed as the union of two statements similar to those in Example 12 (to avoid duplicate results, the UNION statement should be used rather than UNION ALL).

As discussed in later sections, joins in ESL can also be expressed using UDAs, which can offer significant advantages in very special situations.

4.3 Time Ticks

In the next example we generate a delayed stream.

Example 13 Delayed Stream: *List auctions 10 minutes after they were started*

```
SELECTitemID, SellerID, start_price
FROM TickSecond AS T,
     TABLE(OpenAuction OVER 10 MINUTE PRECEDING T) AS D
WHERE CAST(T.time - D.time AS MINUTE) = 10 MINUTE;
```

Here, TickSecond is a stream containing the timestamp generated every second by the system. ESL provides functions that produce ticks at arbitrary intervals, and more complex periodic ticks. Therefore, the TABLE construct just discussed generates a time-varying table from a stream. It has all the properties of a table, and can, e.g., be unioned with another table, but not with a data stream.

5 Base Aggregates: Blocking vs. NonBlocking

ESL supports User Defined Aggregates (UDAs), which are natively defined in SQL and are the basis for the expressive power of the language. ESL adopts from SQL-3 the idea of specifying a new UDA by an INITIALIZE, an ITERATE, and a TERMINATE computation; however, ESL let users express these three computations by a single procedure written in SQL [6]— rather than by three procedures coded in procedural languages as prescribed by SQL-3[6]. Readers who are already familiar with UDAs can skip to the next section.

Example 14 defines an aggregate equivalent to the standard AVG aggregate in SQL. The second line in Example 14 declares a local table, **state**, where the sum

[6]Although UDAs have been left out of SQL:2003 specifications, they were part of early SQL-3 proposals, and supported by some commercial DBMS.

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[7]. Thus, the TERMINATE statements are processed just after all the input tuples have been exhausted. Therefore, the UDA in Example 14, below, is blocking.

Example 14 *Defining the standard aggregate average*

```
AGGREGATE myavg(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 15, below, where the new values are given a higher weight than the old values to assure the exponential decay of their importance.

Example 15 *Continuous Average with exponential decay*

```
AGGREGATE decay_online_avg(Next Real) : Real
{    TABLE state(tsum Real, cnt Int);
     INITIALIZE : {
        INSERT INTO state VALUES (Next, 1);
     }
     ITERATE: {
        UPDATE state
```

[7]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.

```
          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 15 where the TERMINATE state is empty or missing all together, are nonblocking and can be applied freely to data streams. A common implementation for SQL aggregates is that the data are first sorted according to the GROUP-BY attributes: thus the very first operation in the computation is a blocking operation. Instead, ESL uses a (nonblocking) hash-based implementation for the GROUP-BY calls of the UDAs.) This default operational semantics leads to a stream-oriented execution, whereby the input stream is pipelined through the operations specified in the INITIALIZE and ITERATE clauses: the only blocking operations (if any) are those specified in TERMINATE, and these only take place at the end of the computation.

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 6, or with a group-by clause as shown in the Example 16, below.

Example 16 *Find the recent average price of the items offered by each seller.*

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

Observe that **AskdPrice** no longer contain the timestamp column; indeed, SQL semantics only allow the group-by values be returned along with the aggregate columns. However, to return the **start_time** timestamp, we coud modify the definition of our UDA to accept timestamps as the additional input argument and to return it. In general, UDAs provide a very powerful and flexible mechanism for specifying computations, on both database tables and data streams [1].

UDAs make SQL Turing complete on databases, insofar as they can express every function on the database computable by a Turing Machine [4]. Non blocking UDAs are those that have an empty or missing TERMINATE: these make SQL complete for data stream applications insofar as it can express every nonblocking query expressible in any other possible language [4]. These theoretical results on the power of UDAs were reinforced by our concrete experience which proved 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 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** 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**. But, expressed in this naive form the query could require infinite memory. To avoid this, we could use windows (see Section 6). But the use of windows can 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 allow us to compute the duration with bounded memory, given that the number of calls drops exponentially with length. This can be easily done using the following UDA:

Example 17 *List the length of every call in* calls(call_ID, event, time)

```
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 aggregate enhance the performance of UDAs and the user convenience; thus, they are part of ESL, although their functionality could be capture directly using base UDAs. For instance, Example 18, below, shows how the combination of window and slide called 'tumble' [3], can be expressed by a nonblocking base UDA.

Example 18 *Average on a Tumbling Window of 200 Tuples*

```
AGGREGATE tumble_avg(Next Int) : Real
{     TABLE state(tsum Int, cnt Int);
```

```
    INITIALIZE : {
        INSERT INTO state VALUES (Next, 1);
    }
    ITERATE: {          /* part 1 /*
        UPDATE state
            SET tsum=tsum+Next, cnt=cnt+1;
                        /* part 2 */
        INSERT INTO RETURN
            SELECT tsum/cnt FROM state
            WHERE cnt % 200 = 0;
                        /* part 3 */
        INSERT INTO state VALUES (Next,1)
            WHERE cnt % 200 = 1
    }
    TERMINATE : { }
}
```

The tumble definition for averages shown above has been derived from the definition of the base aggregate of Example 14, by basically assembling in its ITERATE statements (i) the ITERATE statements of Example 14, (ii) the TERMINATE statements of Example 14 qualified by the end-of-tumble condition 'WHERE **cnt % 200 = 0**', and (iii) the INITIALIZE statements of Example 14 qualified by the condition 'WHERE **cnt % 200 = 1**', which denotes the start of a new tumble. Therefore, an efficient nonblocking implementation of the tumble version of a UDA can be derived directly from its blocking counterpart. As discussed in Section 6.2, the ESL compiler takes advantage of this observation to implement tumble UDAs efficiently.

6 Window Aggregates

While each window UDA could be defined as a base UDA, ESL uses specialized constructs for window UDAs to achieve better performance, user convenience and compatibility with SQL:2003 standards. Following these standards, the OVER clause is used to specify (i) the type of window (i.e., logical or physical), (ii) the size of the window (using time or tuple count, resp.), and (iii) the columns in the partition-by clause (if any). When applied to data streams, however, the ORDER BY clause should be omitted, since it is always the case that data streams are ordered by their timestamps [8].

In Example 19, below, we specify that a physical window with a size of 10 items (ROWS 9 PRECEDING), should be maintained for each seller (PARTITION

[8]The FOLLOWING clause is not supported in the current version of ESL.

BY **sellerID**), to compute the average price of the 10 items in the window. Therefore, ESL follows SQL:2003 in requiring that PARTITION BY must be used with window aggregates, while GROUP BY is used with based aggregates (although the two constructs are semantically very similar).

Example 19 *Count-Based Window: For each buyer, maintain the average selling price over the last 10 items sold.*

```
CREATE STREAM LastTenAvg
    SELECT sellerID,
            AVG(price) OVER(PARTITION BY sellerID ROWS 9 PRECEDING),
            current_time
    FROM ClosedPrice;
```

In the next example, we return to the user the current bid and the highest bid during the last 10 minutes (logical window).

Example 20 *UDA on time-based window: highest bid in the last 10 minutes.*

```
INSERT INTO stdout
SELECT  current_time,
        max(bid_price) OVER (RANGE 10 MINUTE PRECEDING)
FROM Bid
```

In this example, we specify a duration-based size on the stream **Bid** which has only latent timestamps. To support this request, ESL generates a timestamp as tuples are added to the window, using the **current_time** function—thus the semantics for tuples with explicit timestamps is used.

This example illustrates the situation where latent timestamps are instantiated lazily, when needed: most query operators, e.g., selection, projection and base aggregates, do not require latent timestamps to be instantiated at all.

The third situation is that of UNLIMITED PRECEDING windows which was illustrated by Example 6.

While in SQL:2003 windows are only supported on built-in aggregates, in ESL windows are supported on general UDAs. As discussed in Section 6.2, ESL also supports the window features called *slides* and *tumbles* that have gained wide acceptance in data stream languages [3]. We will next discuss the simple rules that integrates these complex features and link window aggregates to their basic counterparts.

6.1 Defining UDAs on Windows

ESL assures very efficient support for window aggregates by integrating (i) the definition of their base counterparts, with (ii) window-specific improvements specified by the user.

Take for instance Example 6, where RANGE UNLIMITED PRECEDING specifies the continuous version of **max**. The standard, blocking version of **max** can be specified as follows:

Example 21 *Defining the base max UDA*

```
AGGREGATE max(Next Real) : Real
{ TABLE memo(cmax Real);
  INITIALIZE : { INSERT INTO memo VALUES (Next)
       }
  ITERATE: { UPDATE memo SET cmax= Next WHERE Next > cmax
       }
  TERMINATE: { INSERT INTO RETURN SELECT cmax FROM memo
       }
}
```

We have here defined the SQL-2 max aggregate that is blocking insomuch as it returns its results in the TERMINATE statement. However a continuous, nonblocking version of this aggregate can simply be derived by moving the statements now found in TERMINATE into, and at the end of, the ITERATE state. Actually, the same statements must also be appended to those in INITIALIZE state since we also want the result to be returned to for the first tuple. An equivalent approach consists in keeping the original definition, but then executing the the TERMINATE statements after the INITIALIZE or ITERATE statements are executed. These two approaches define the equivalent abstract and concrete semantics used by ESL for all UDAs called OVER windows with the RANGE UNLIMITED PRECEDING option.

While the efficient implementation of UDAs over unlimited windows can be derived directly from the base case, the optimization UDAs called over logical or physical windows poses a challenge that the user must address for each specific UDAs taking advantage of special constructs made available for this purpose by ESL. These special constructs are

- Special CREATE WINDOW AGGREGATE declarations whereby the user can specify an optimized implementation using the **inwindow** and EXPIRE construct,
- the **inwindow** construct whereby the user can access the values stored in the window, which the system manages for performance reasons, and

- the EXPIRE construct to identify expiring tuples and to perform delta computation for maintaining the aggregate.

The first two constructs are illustrated in the Example 22, below, describing a naive implementation of **max** over a finite size window.

Example 22 *A Naive Version of* **max** *on a finite window*

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

The declaration TABLE **inwindow(wnext** Real) instructs the system to store the input values in a special window buffer that will be called **invindow**[9]. Besides the efficiency due to the direct implementation by the system, **invidow** offers the significant benefit of unifying the treatment of logical and physical windows by shielding the user from low-level implementation details needed to support the two different kinds of windows. As a result the user can specify one single implementation that will work efficiently for both logical and physical windows. However, although Example 22 achieves this unification, it also leaves much room for improvements in terms of performance, as discussed next.

A first observation to be made about Example 22 is that our window version of MAX calls on the base **max** aggregate for tables; but this is not a recursive call, since the two aggregates are internally treated as two different procedures.

The second observation is that a similar derivation of the window aggregate from its corresponding base aggregate can be made trivially for any base UDA, as it was done for **max**. Therefore, ESL uses this approach to derive a default definition for any base aggregate called on a window, for which there is no specialized window declaration.

The third observation is that this default definition is often much less efficient than one using the EXPIRE construct discussed next.

For instance, Example 23 defines a highly optimized implementation of AVG. Thus, in our UDA definition, we have a fourth state called EXPIRE whereby the effect of expired tuples can be used to perform delta-maintenance on the window

[9]The names of the columns of **invindow** can chosen by the user: the name of the table itself and data types of the columns cannot.

UDA. For AVG, the delta maintenance consists in decreasing the sum by the value of the expired tuple and the count by 1. The the result is the same whether this delta computation is performed as soon as a new 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.

The system also provides the user with the predicate **oldest** to identify the oldest tuple in the buffer (the front of the queue), thus providing the convenience of processing expired tuples one-by-one.

Example 23 *The New Construct* EXPIRE

```
 WINDOW AGGREGATE myavg(Next Real) : Real
 {  TABLE state(tsum Int, cnt Real);
    TABLE inwindow(wnext Real);
    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;
          }
    EXPIRE: { /*when there are expired tuples take the oldest */
       UPDATE state SET cnt= cnt-1,
                  tsum = tsum - oldest() → tsum
 }
```

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**. Thus, **oldest()** is built-in function that delivers the oldest among the tuples in **inwindow**: thus for **oldest()** → **tsum** delivers the **tsum** column in this tuple.

Upon the arrival of a new tuple, the system first proceeds at executing any outstanding EXPIRE event. Therefore, EXPIRE can be executed at any point of the closed time interval that starts when a tuple actually expires and ends when the next input tuple is actually detected—whereby the ITERATE statements are next executed on this tuple.

The use of EXPIRE in UDAs represents a significant improvement over the delta maintenance constructs presented in [7], since it produces shorter definitions and a more efficient implementation.

Tuples in **inwindow** can be deleted and updated by users (but not inserted). For instance, in computing a **sumdistinct** aggregate, when a new tuple comes in we eliminate older duplicate values from the window.

Example 24 *SumDistinct with windows*

```
WINDOW AGGREGATE sumdistinct (Next Real) : Real
{ TABLE thesum(tsum real);
  TABLE inwindow(wnext real);
  INITIALIZE : {
    INSERT INTO thesum 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*

```
WINDOW AGGREGATE max (Next Real) : Real
{ TABLE inwindow(wnext real);
  INITIALIZE : { } /* 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*/
}
```

These examples exemplify the desirability of customizing the delta maintenance to be performed 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.

6.2 Slides and Tumbles

The *slide construct* represents a very useful extension to the SQL:2003 standards that is often used in DSMSs [3]. ESL's support for this construct is not limited to built-in aggregates but extends to arbitrary UDAs.

For a first example, consider the situation where, every 2 minutes, we want to see the max bid made in the previous ten minutes. Then, we can simply write:

Example 26 *UDA on time-based window: Every 2 minutes, print the highest bid in the recent 10 minutes.*

```
INSERT INTO stdout
   SELECT max(bid_price, itemID)
         OVER (RANGE 10 MINUTE PRECEDING SLIDE 2 MINUTE)
   FROM Bid
```

In this example, we have a window of size $W = 10$ minutes with a slide of size $S = 2$ minutes. Therefore we have that $S < W$. (We also have that W is a multiple of S; this represents a typical situation, but not a requirement enforced by ESL.) The treatment of slides and windows depends on whether $S < W$, as in this case, or $S \geq W$. We will first discuss the second case and return to the first case after that.

Situations where $S \geq W$ are called *tumbles*. The ESL compiler implements window UDAs on tumbles by extrapolating from their base definitions. For instance, Example 18 proposes an implementation for tumble **avg** based on the definition of the base **avg** given in Example 14. In fact, rather than actually using the rewriting shown in Example 18, the ESL system retains the original definition in Example 14, but then, besides the ITERATE clause it executes the TERMINATE clause and the INITIALIZE clause, respectively, when a tumble ends and a new one begins. For physical windows the old tumble ends when the count of the input tuples is a multiple of S, i.e., it is equal to $n \times S$ for some positive integer n. Then the next tumble begins at $n \times S + 1 + S - W$. For logical windows, S and W denote actual time spans; then, the TERMINATE clause (resp. the INITIALIZE clause) is executed when the first tuple with timestamp greater or equal to $n \times S$ (resp. $n \times S + 1 + S - W$) arrives.

Let us consider now the case where $S < W$ and therefore, there is no tumble. In this case, the slide is supported by a simple modification of the window version of the UDA (i.e., that supplied by the user, and if this is not available the default definition, such as that of Example 22). Said modification consists in executing the INSERT INTO RETURN for input tuples only when the *on-slide* condition is true. For physical windows the on-slide condition is true for each tuple that is a multiple of the slide. For logical windows, we can view the n-th slide as a time interval, closed to the left and open to the right, that has end points $n \times S$ and $(n+1) \times S$. Then, the on-slide condition is true for the first arriving tuple in this interval (thus, no on-slide condition occurs for slides during which no tuples arrive).

Associative Aggregates A significant optimization is possible for associative aggregates [10] These UDAs can be processed by first creating continuous tumbles for the slides, and then calling the standard window aggregate for each result returned by the slide. Thus, for query in Example 26 we can create tumbles of two minutes and then process the results they produce every two minutes with the UDA of Example 25. In other words, we can recast the query of Example 26 into the following pair of queries:

Example 27 *Optimizing Windows with Slides*

```
CREATE STREAM TumbleBids
SELECT max(bid_price, itemID) AS twominutes
        OVER (RANGE 2 MINUTE PRECEDING SLIDE 2 MINUTE)
FROM Bid;

INSERT INTO stdout
SELECT max(bid_price, itemID)
        OVER (RANGE 10 MINUTE PRECEDING)
FROM TumbleBid;
```

Obviously, this approach reduces both the storage and computation cost required by the window UDA. Associative UDAs must be identified by the key word ASSOCIATIVE added to their declaration. Thus the declaration of MAX in Example 25 should be as follows:

Example 28 *Max with windows, with slide optimization*

```
ASSOCIATIVE WINDOW AGGREGATE max (Next Real) : Real
   { ... body as in Example 25 * }
```

Not every UDA is associative: for instance the distinct version of sum defined in Example 24 is not associative and it is therefore impervious to slide optimization. On the other hand, some aggregate that are not associative can be easily recast into forms that enable this optimization. For instance, AVG can be recast as follows:

[10] A binary function F is associative if bracketing has no effect on the value returned by F: i.e., if $F(x, F(y, z)) = F(F(x, y), z)$, for all x, y, z values in the domain of F.

Example 29 *Recasting AVG into sum*

```
WINDOW AGGREGATE RECAST avg(Next) AS sum(Next)/sum(1);
```

The ESL compiler uses this statement to rewrite a call to the window version of AVG into a call to a pair of window sum aggregate (where the window parameters remain the same).
The next example shows an alternative definition of AVG, which is now recast into a new base UDAs.

Example 30 *Recasting AVG into ASSCAVG*

```
WINDOW AGGREGATE RECAST avg(Next) AS asscavg(Next, 1);

ASSOCIATIVE WINDOW AGGREGATE
           assavg(Nextsum Real, Nextcount Int) : Real
{  TABLE state(tsum Real, tcnt Int);
   TABLE inwindow(wnext Real, wcountInt) }
   INITIALIZE : {
      INSERT INTO state VALUES (nextsum, nextcount);
         }
   ITERATE : {
      UPDATE state SET tsum=tsum+nextsum, cnt=cnt+nextcount;
      INSERT INTO RETURN
         SELECT tsum/cnt FROM state;
         }
   EXPIRE: {  /* when there are expired tuples take the oldest */
      UPDATE state SET tcnt= tcnt-wcount,
                 tsum = tsum - (select wnext FROM inwindow
                                          WHERE oldest(inwindow)) }
}

           /* the definition of regular version of asscavg */
   AGGREGATE asscavg(nextsum Real, nextcount Int) : (Real, Int)
   {TABLE state(tsum Real, tcnt Int);
   INITIALIZE : {
         INSERT INTO state VALUES (nextsum, nextcount);
   ITERATE : {
         UPDATE state SET tsum=tsum+nextsum, tcnt=tcnt+nextcount }
   TERMINATE : {
      INSERT INTO RETURN
            SELECT (tsum, tcnt) FROM state }
   }
```

In these sections, we have discussed various techniques made available to the user at the logical level; the various techniques used to optimize window implementation at the physical level (e.g., when main memory is limited) will be discussed in future reports.

7 Conclusion

The two main features that set ESL apart from other SQL-based languages for data streams are (i) the restriction that only one data stream is allowed in the FROM clause of an ESL query, and (ii) the ability of defining powerful UDAs (both base UDAs and window UDAs) in the language itself.

In ESL, the only binary query operators on data streams is UNION. All the remaining queries define unary operators by allowing only one data stream in their FROM clauses. The continuous queries so defined have a simple semantics, as described next.

1. Non blocking ESL statements containing selection, projection and UDAs on a single stream produce exactly the same result as they were posed on database tables. Indeed, if Q is such a query, and S_t denotes the stream input up to time t, then the cumulative result of applying Q to S_t until time t is the same as applying Q to the table S_t: i.e., $Q(S_t)$.

2. Also, because of monotonicity, the semantic our query processor can be defined by describing its response when new delta tuples arriving at t, and deriving its cumulative response for that. For the selection, projection and UDA queries, the response is simply $Q(S_t) \backslash Q(S_t^-)$, where S_t^- denotes the tuples in the stream until time t, excluded.

3. In now we have joins with the database table, we define the incremental response at time t, as the join of the delta tuples with the snapshot of the tables at time t.

4. Also well-defined is the meaning of materialized window views, and table functions (i.e., those defined using the OVER construct). Basically, they are viewed as time-varying DB tables–therefore the meaning of joining data streams with these is well defined. (This also addresses the issue of stream joins.)

Out approach to semantics does not deal directly with the semantics of joins. However, the functionality of joins was not lost since ESL supports table-like concrete views on data streams, and joins of data streams with tables.

ESL also supports ad hoc queries on database tables and concrete views.

ESL achieves superior expressive power by using UDAs natively defined in ESL itself, including base aggregates, such as those of SQL-2, and OLAP aggregate functions operating on windows. In fact, ESL provides a very effective framework whereby the two kinds of aggregates are fully integrated and new features, such as tumbles and slides, are efficiently supported for arbitrary UDAs, not just built-in aggregates.

References

[1] Atlas user manual. http://wis.cs.ucla.edu/atlas.

[2] Haixun Wang Carlo Zaniolo, Richard Luo. The streammill client user manual. 2004.

[3] 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.

[4] Yan Nei Law and Haixun Wang adn Carlo Zaniolo. Query languages and data models for database sequences and data streams. In *VLDB 2004, Proceedings of 30th Int. Conference on Very Large Data Bases*. Morgan Kaufmann, 2004.

[5] 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.

[6] Haixun Wang and Carlo Zaniolo. Using SQL to build new aggregates and extenders for object-relational systems. In *VLDB*, 2000.

[7] Haixun Wang, Carlo Zaniolo, and Chang R. Luo. ATLaS: a turing-complete extension of sql for data mining applications and streams— VLDB 2003 demo. http://wis.cs.ucla.edu/publications.html.