A Data Stream Language and System Designed for Power and Extensibility

Yijian Bai
UCLA
bai@cs.ucla.edu

Hetal Thakkar
UCLA
hthakkar@cs.ucla.edu

Chang Luo
UCLA
lc@cs.ucla.edu

Haixun Wang
IBM T.J. Watson R. C.
haixun@us.ibm.com

Carlo Zaniolo
UCLA
zaniolo@cs.ucla.edu

ABSTRACT

By providing an integrated and optimized support for user-defined aggregates (UDAs), data stream management systems (DSMS) can achieve superior power and generality while preserving compatibility with current SQL standards. This is demonstrated by the Stream Mill system that, through is Expressive Stream Language (ESL), efficiently supports a wide range of applications—including very advanced ones such as data stream mining, streaming XML processing, time-series queries, and RFID event processing. ESL supports physical and logical windows (with optional slides and tumbles) on both built-in aggregates and UDAs, using a simple framework that applies uniformly to both aggregate functions written in an external procedural languages and those natively written in ESL. The constructs introduced in ESL extend the power and generality of DSMS, and are conducive to UDA-specific optimization and efficient execution as demonstrated by several experiments.

Categories and Subject Descriptors: H.2.3 Languages: Query languages

General Terms: Languages

Keywords: Data Stream, Query Language, Data Stream Management System

1. INTRODUCTION

Data Stream Management Systems (DSMSs) and their languages represent a vibrant area of database research [4, 21]. However, the view that this burgeoning research area naturally belongs to databases is not shared by everyone. Indeed, researchers from other areas are quick to point out that (i) the requirements and enabling technology for processing data received on the wire are very different from those of data stored on disk, and (ii) database languages, such as SQL which had a mixed success with advanced database applications, lack the power and flexibility needed to support the range and complexity of applications encountered on data streams. Admittedly, extending the power and flexibility of relational query languages has provided an arduous research challenge for the last 20 years, leading to major additions such as recursive queries and object-oriented extensions. However, even after those significant extensions, there remain important data-intensive applications, such as data mining and sequence queries, that are not supported well by SQL. SQL's expressivity problems become even more serious when expressing continuous queries on data streams. Since blocking operators cannot be used in continuous data stream queries [4], non-monotonic operators, such as traditional aggregates and the EXCEPT construct (expressing set difference) are disallowed [24]. Moreover, if we eliminate from SQL its non-monotonic operators (such as EXCEPT), some of its monotonic queries are nolonger expressible. Thus SQL is not complete w.r.t. non-blocking queries (and neither is relational algebra) [24].

Moreover, the traditional remedy of embedding SQL queries in a procedural programming language, where the more complex computations can be easily expressed, is no longer effective in the push-based environment of data streams. Indeed, DSMSs must operate by taking tuples from input buffers and continuously pushing the query results to the output buffers—without waiting for get-next requests from an embedding PL as traditional databases do. Finally, extensions based on data-blades, whereby large 'blobs' are passed to external functions, do not dovetail with the incremental computation required in continuous queries.

Given the daunting expressive power limitations of SQL and other traditional query languages on data streams [24], our first objective is to find extensions that can overcome SQL's limitations and turn it into a powerful continuous-query language. Stream Mill has solved this difficult problem by building on (i) window constructs, and (ii) user-defined aggregates (UDAs).

UDAs natively defined in SQL were shown in [30] to be effective at supporting data mining functions and other complex queries not expressible efficiently in SQL:2003. Furthermore, in [24] it was shown that SQL with UDAs is Turing-complete—and also complete for continuous queries, inasmuch as monotonic SQL operators with monotonic UDAs can express all possible non-blocking queries. The theoretical results of [24] underscored the importance of UDAs, but did not address the actual language design and implementation issues, such as: how to best define UDAs on logical and physical windows (with or w/o slides), optimize their computation, and integrate native and imported UDAs. These issues are addressed in this paper, which makes the following contributions to the state of the art:

1. A fully integrated syntax and semantics for the various window constructs (i.e. logical and physical windows combined with slides, tumbles, unlimited preceding, and active expiration modifiers), applied to arbitrary UDAs that can be defined in either native SQL or in an external procedure language. This integration makes our UDAs powerful to express data stream queries.
2. Several implementation and optimization techniques used in the Stream Mill DSMS system to support the integration of the window constructs and arbitrary UDAs (e.g., the EXPIRE constructs are specially used for delta computations, and the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CIKM'06, November 5–11, 2006, Arlington, Virginia, USA.
Copyright 2006 ACM 1-59593-433-2/06/0011 ...$5.00.

pane-based slide optimization [17]).

3. The effectiveness of our window-UDA optimization techniques is validated via several experiments. The experiments also show that UDAs defined in native SQL only incur a small performance penalty compared to those implemented in a procedural language (typically a 10-20% slowdown), with significant advantage in conciseness and convenience.

The UDAs and window constructs of ESL represent minimal extensions w.r.t. SQL:2003 standards. Yet they have made possible a number of sophisticated applications. In particular:

- Data stream mining applications can be easily written using window UDAs [25],
- Parallel Finite State Automata (FSA) for processing SAX streams can be implemented efficiently using UDAs, thus enabling unification of relational and XML streams [33],
- Approximate computations, time series analysis, and RFID applications can also be expressed using UDAs, as discussed in Section 7.
- Active expiration window semantics [23, 22] and concrete views can also be supported as discussed in Section 5.

Note that, while [25], [33] and [6] each has applied the ESL language on different application domains, in this paper we instead present the constructs and techniques used to implement the ESL language itself.

Short Overview In the next section, we cover the use of SQL constructs to express basic continuous queries on data streams. ESL also supports ad hoc queries on stored database tables and on virtual tables derived from data streams via concrete views and table functions. We omit their discussion here to save space and to concentrate on the treatment of UDAs and different types of windows over UDAs, which represent the cornerstone of ESL's query power. UDAs are introduced in Section 3, which provides a simple syntactic characterization of non-blocking UDAs versus blocking UDAs. Window UDAs are introduced in Section 4, along with constructs and techniques for delta-maintenance that can be used on different kinds of windows and aggregates. Section 5 discusses new types of windows based on slides and active expiration, and implementation of unique features of ESL. In Section 6, we present experiments to evaluate the performance of our new constructs. In Section 7, we show that ESL's expressive power enables it to express complex applications concisely and efficiently. These applications are difficult or even impossible to express without window UDAs.

2. CONTINUOUS QUERIES

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 [8, 4, 21, 24]. In the Stream Mill system, data streams are declared using a CREATE STREAM clause. In addition to the data stream schema, the 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 timestamp values are contained in the arriving tuples, and are specified with an ORDER BY clause. For instace, the data stream **OpenAuction** in Example 1 is declared as having **start_time** as its external timestamp. Instead, **ClosedAuction** is declared as having internal timestamp, which is denoted by using the reserved column name, **current_time**. In this case, the system generates a new timestamp upon arrival of the tuple and inserts it in the arriving tuple. External and internal timestamps will be called *explicit*: ESL operators treat all explicit timestamps in the same way. The stream **Bid** in Example 1 does not have an associated timestamp, denoted by absence of the ORDER BY clause. Timestamp values consistent with tuple order are dynamically generated for these tuples, when they are used in operators that have timestamp dependent semantics. Thus, these streams are referred to as having *latent* timestamps.

Externally timestamped streams may receive out-of-order tuples due to reasons, such as network latency. To ensure tuple timstamp order in external streams, these out-of-order tuples are put in a separate stream. These tuples can either be discarded or re-merged with the original stream in the correct order using a mechanism similar to the *slack* mechanism proposed by Aurora system [14].

The source clauses in Example 1 specify that the streams are going to be imported from the specified ports. Further discussion of stream importation is omitted due to space limitations.

EXAMPLE 1. *Declaring Streams in ESL*

```
CREATE STREAM OpenAuction (itemID int,
        sellerID char(10), price real, start_time timestamp)
    ORDER BY start_time; /* external timestamps */
    SOURCE 'port4445';
CREATE STREAM ClosedAuction(itemID int,
        buyerID char(10), price real, current_time timestamp)
    ORDER BY current_time; /* internal timestamp */
    SOURCE 'port4446';
CREATE STREAM Bid(itemID int,
        price real, bidderID char(10), bid_time timestamp)
    SOURCE 'port4447';
```

2.1 Single Stream Transducers

In ESL it is easy to write continuous queries that act as single stream transducers. 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, price, start_time
    FROM OpenAuction WHERE 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 in increasing order of timestamp values. 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 is similar to the CREATE VIEW mechanism in SQL. For instance, Example 3, defines a data stream that is basically the same as the data stream delivered to the user in Example 2. However, instead of being delivered to the user, **expensiveItems** is now a data stream that can be used by other queries.

EXAMPLE 3. *Deriving a New Data Stream*

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

Aggregates: Aggregates are the final construct that can be applied to an individual data stream. Example 4 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 streams, the ESL compiler also checks that **decay_online_avg** is a non-blocking UDA—a property that is easily

inferable from the syntactic structure of the UDA definition, as discussed in Section 3.

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 5, below, shows the use of an unbounded window, whereby the **min** returns the lowest start price seen so far.

EXAMPLE 4. *The Recent Average of the Closing Bids*

```
SELECT decay_online_avg(price) FROM ClosedAuction
```

EXAMPLE 5. *The Smallest Asking Price, For Each Seller*

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

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. 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. UDAs invoked without a window modifier will be called *base aggregates* as in Example 4[1], whereas UDAs invoked with a window modifier will be called *window aggregates* as in Example 5.

All basic SQL:2 aggregates and their window versions are supported as builtins in ESL. In addition, ESL supports efficiently window constructs on arbitrary UDAs—not just builtin ones. This feature provides much greater power and flexibility than those provided by other DSMSs or commercial implementations, and will be discussed in Section 4.

Queries Spanning Data Streams and DB Tables: ESL also supports the join of a data stream with database relations. In such joins, the FROM clause of an ESL query can contain any number of database tables but only one data stream (or the window join of several data streams, producing a single data stream).

2.2 Union

The UNION operator is directly applicable to multiple data streams without window modifiers. (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 6, below, sort-merges the **OpenAuction** and the **ClosedAuction** streams to select the start price and the final price for each item:

EXAMPLE 6. *Price History Query*

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

The union of data streams with explicit timestamps, produces a stream that is ordered by its timestamp. Thus, the union is implemented as a sort-merge operation that assures that the output tuples are sorted by their timestamps. Therefore, when computing union of several streams and none of their buffers are empty, we select the tuple with the minimum timestamp. If the buffer of any stream is empty, we use an advanced heartbeat mechanism to unblock the operator, as discussed in [7].

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.

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

3. BASE AGGREGATES

ESL supports UDAs as proposed in [30]. The idea of defining a new UDA by specifying the computation to be performed in the three different states called INITIALIZE, ITERATE, and TERMINATE, is also advocated by [1, 14]. This approach is implemented in several DBMSs and DSMSs, which normally require such computation to be specified in an external PL; but in ESL they can also be specified natively using ESL itself [30]. In our examples, we will use UDAs of this second type, since they represent a unique feature of ESL and they produce code that is clear, concise, easy to write. Here, we briefly discuss definition of such UDAs to allow better understanding of advanced concepts discussed in latter sections. Reader is referred to [30] for more detailed discussion on UDAs.

For instance, Example 7 defines a UDA equivalent to the standard AVG aggregate in SQL. The second line in Example 7 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. The INITIALIZE clause 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[2]. Since the TERMINATE statements are processed just after all the input tuples have been exhausted, the UDA in Example 7 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 7. *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 8, below, where the new values are given a higher weight than the old values to assure exponential decay of their importance.

EXAMPLE 8. *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; }
}
```

UDAs, such as those of Example 8 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

[2]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.

pipelined through the operations specified in the INITIALIZE and ITERATE clauses.

The UDAs defined so far will be called *base* aggregates. Base aggregates are called using a group-by clause as in Example 9 below, or without a group-by clause as in Example 4.

EXAMPLE 9. *Find the Recent Average Price of the Items Offered by Each Seller.*

```
CREATE STREAM AskdPrice AS
    SELECT sellerID, decay_online_avg(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 [24]. 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 [24]. 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.

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 only be applied on data streams through the window constructs, discussed in the next section. In theory, window aggregates are not needed for completeness, since they could be expressed via complex base UDAs. However, they provide major benefits in terms of user-convenience and performance, as shown in the experiments. Thus, window aggregates represent a critical (and highly optimized) construct in ESL.

For instance, Example 10 below shows the definition of an aggregate that behaves as the **unbounded preceding** version of average.

EXAMPLE 10. *The Cumulative Average–i.e., its 'unbounded 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 7, by taking the return clause from TERMINATE and appending it to INITIALIZE and ITERATE clauses. 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 the 'unbounded preceding' versions of UDAs [3]. For other types of windows, however, the UDA implementation that can be derived from its base definition tends to be inefficient. Better solutions are indeed available, as discussed in the next section.

4. WINDOW AGGREGATES

ESL optimizes window aggregates both at the physical and logical level through:

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

The use of the CREATE WINDOW AGGREGATE declaration and the **inwindow** construct are illustrated by Example 11 below, which defines a naive implementation of **avg** over a finite-sized window.

EXAMPLE 11. *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 11, 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**[4]. Incoming tuples (expiring tuples) are automatically added to (deleted from) **inwindow** by the system 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.

Although the use of **inwindow** assures efficient memory management, the algorithm shown in Example 11 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 introduces the EXPIRE construct, discussed next, which enables the above computation in time $O(n)$. Example 12 defines a highly optimized implementation of AVG, using the EXPIRE state 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 the next tuple comes in, or anywhere in between these two instants (See 5.4 for more details).

EXAMPLE 12. *The New* EXPIRE *Construct*

```
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

[3]In fact, instead of rewriting the UDA, ESL simply executes the TERMINATE statement after INITIALIZE or ITERATE execution for each incoming tuple

[4]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.

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 just **oldest()**, i.e. without the column name.

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 of Example 12, ESL uses the definition of Example 11 to support the application of **avg** over finite windows, and for 'unbounded preceding' windows ESL instead uses the definition in Example 10. 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 'unbounded preceding' and the 'finite window' versions can be trivially derived by simple syntactic rewriting.

The table **inwindow** is managed by the system, which inserts new tuples and deletes expiring tuples according to the window type (i.e., logical or physical) and the range of the window. Insertion of new tuples in the window by ESL statements is not allowed since it is incompatible with the window semantics. 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. This feature allows us to write an improved version of the **max** aggregate, as shown in Example 13, instead of keeping all tuples inside the window range, for each incoming tuple we can eliminate older tuples of less or equal value, as they cannot be the max value in the current window or any future window. Thus the oldest tuple in the window has the max value, which is therefore returned as the result of the aggregate.

EXAMPLE 13. *MAX with Windows*

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

The **max** aggregate, although simple, can be a very useful construct. For instance, many applications require sampling from a moving window over streaming data. Babcock et al [5] proposed an algorithm known as "priority-sample". As a new element arrives, it is assigned a random priority between 0 and 1. An element is put into the sample if it has the highest priority. To maintain a sample of size k, we generate k random priorities for each tuple.

Clearly, priority-sample of size 1 can be implemented by applying the **max** aggregate on the random priorities. To implement priority-sample of size k, we can modify the **max** aggregate so that it handles k priorities. As for the memory requirement, it can be shown that for aggregate **max**, the expected number of records that need to be kept in **inwindow** is $H(n)$, the nth harmonic number, where n is the size of the window. Therefore, for a sample of size k, both the expected and high-probability upper bound of **inwindow** size is $O(k \log n)$ [5].

Thus, the optimized treatment of window aggregates finds many natural applications and provides opportunities to support advanced constructs such as slide and tumble, as we discuss next.

5. BEYOND SQL:2003 WINDOWS

The logical and physical windows discussed so far, are already found in SQL:2003 for built-in OLAP functions and are also supported in many commercial DBMSs. In this section we extend that framework to deal with concepts, such as slides and tumbles, that have proven useful in many data stream applications, and are supported on built-in aggregates by many DSMSs. Stream Mill extends this framework to allow slide and tumble specification on arbitrary UDAs.

5.1 Slides and Tumbles

SLIDE is an important construct supported by many DSMSs on built-in aggregates, whereby aggregates return results every so often, rather than after each processed tuple, providing significant opportunities for query optimization [17]. For instance, to find the sum of bids for each item, over the last 50 bids with slide 10 bids, we can use the following ESL statement. Note 49 preceding stands for the current tuple and the 49 previous tuples, thus total of 50 tuples.

EXAMPLE 14. *Use of the SLIDE Construct*

```
SELECT itemID, sum(bid_price)
    OVER(PARTITION BY itemID
        ROWS 49 PRECEDING SLIDE 10)
FROM Bid
```

When the specified slide size is smaller than the window size, we refer to this as a pane window, as in Example 14. On the other hand, when the slide is larger than or equal to the window size, it is called a TUMBLE [16]. ESL provides efficient support for both pane windows and tumbles. The semantics of windows with panes or tumbles can be defined by subdividing the complete history of the stream into slots, where slot size is dependent on both window size and pane size. A slot's yield tuple is defined as the last tuple of the slot. The semantics of window aggregates with slide is that of window aggregates without slide, once all the tuples but those produced by its yield tuples have been dropped from the output stream. This uniform definition of abstract semantics leaves much room for optimization upon implementation, which is very different for tumbles and pane windows. First, consider an example of a tumble as below:

```
ROWS 9 PRECEDING SLIDE 28
```

For tumbles the size of the slot is always equal to the slide, i.e. 28 rows for the above example. Here, we can bypass computation for the first 18 rows in each slot, altogether. In fact, the ESL compiler implements this window call on an arbitrary UDA, by repeating the following operations on each 28 row slot: (i) ignore the first 18 input tuples, (ii) apply the base UDA to the next 10 rows, and (iii) execute its TERMINATE statements when its yield tuple arrives. Observe that when the slide and window are equal, then the yield tuple that prompts the execution of the TERMINATE statement, also causes the aggregate computation to be restarted for the next slot; thus the next incoming tuple will be processed by the INITIALIZE statement of the restarted UDA, i.e., the data stream is partitioned into exclusive slots each processed as a tumble.

The optimization of pane windows for aggregates based on their algebraic properties was discussed in [17], along with their significant benefits in terms of computation and memory savings. Although, compilation techniques for optimizing arbitrary UDAs was not discussed in [17]. Here, we propose a simple framework where this optimization can be applied to arbitrary "algebraic" [5] UDAs [18]. Stream Mill optimizes the computation of such aggregates by

[5]This includes both *distributed* and *algebraic* aggregates as defined in [18]

(i) computing the aggregate on tumbles of size equal to the GCD of slide and window size [17], and (ii) feeding the results into the window version of the same aggregate without any slide. Thus, Example 14 is optimized by rewriting it as follows:

EXAMPLE 15. *Rewrite for Pane and Delta Computation*

```
CREATE STREAM temp AS (
    SELECT itemID, sum(bid_price) AS s
      OVER(PARTITION BY itemID ROWS 9
         PRECEDING SLIDE 10)
    FROM Bid);
SELECT itemID, sum(s)
    OVER(PARTITION BY itemID ROWS 4 PRECEDING)
FROM temp
```

Thus the implementation of a pane window aggregate is performed by cascading its tumble version with its pane-less window version— this simple rewriting technique achieves both the pane optimization and delta optimization described in [17] for arbitrary UDAs, which are defined using the keyword ALGEBRAIC (Example 16). Similar optimization can also be applied in the case of logical windows. We have used this approach in various applications to optimize variance and other statistical aggregates, string-concatenation aggregates, and basic aggregates on window samples. ESL also provides simple constructs whereby, say, the aggregate **myavg** can be rewritten to one returning both count and sum, thus making the rewritten aggregate amenable to this composition-based optimization. Note that without this rewriting, the above cascading is not possible for the **avg** aggregate because of argument mismatch.

When an aggregate is not declared with the keyword ALGEBRAIC, Stream Mill falls back to the standard window aggregate, but ignores its INSERT INTO RETURN statements, for all input tuples but the yield tuples. Indeed holistic UDAs, such as median, cannot be optimized using the composition-based preprocessing, although their delta maintenance and tumble optimization is unaffected by this.

5.2 Active Window Expiration

In SQL:2003, window aggregates produce results only when a new tuple arrives. With the SLIDE construct, we have provided a mechanism to shed some of those results. On the other hand, active expiration produces additional tuples, which are generated when an old tuple expires out of a time-based window [6] [23, 22]. This semantics for logical windows has been recently advocated by several researchers, and it is in fact uniquely apt to enabling certain kinds of new applications, such as RFID-data processing— (see Example 18 later). ESL supports this extension by allowing an ACTIVE EXPIRATION clause to replace the slide clause in the window aggregate invocation. Moreover, the EXPIRE clause of the window aggregate must be expanded with an INSERT INTO RETURN statement that specifies what needs to be returned when a tuple expires. For instance, the EXPIRE statement for the avg aggregate would become:

```
EXPIRE: { /*when there are expired tuples take the oldest*/
    UPDATE state
      SET cnt= cnt-1, tsum = tsum - oldest().wnext;
    INSERT INTO RETURN
      SELECT tsum/cnt FROM state; }
```

Example 13 can be expanded in similar fashion; we then obtain Example 16, below, that also contains the ALGEBRAIC declaration that will trigger its pane-based optimization when it is called with the slide option rather than the active-expiration option:

[6]Only the case of logical windows is of interest since, for physical windows, tuples only expire when a new tuple comes in

EXAMPLE 16. *MAX Extended for Slides and Active Expiration*

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

5.3 Syntax and Semantics

```
window_function ::= aggregate_function OVER <window_spec>
window_spec ::= ( [<window_partition_clause>]
            <window_frame_clause> [<slide_clause>])
window_partition_clause ::= PARTITION BY <column_list>
window_frame_clause ::= <window_frame_units> <window_frame_start>
window_frame_units ::= ROWS | RANGE
window_frame_start ::= UNBOUNDED PRECEDING
              | <window_frame_preceding>
window_frame_preceding ::= <value_spec> PRECEDING
value_spec ::= INTEGER [UNITS]
slide_clause ::= SLIDE value_spec | ACTIVE EXPIRATION
```

Figure 1: Window Aggregate Invocation BNF Grammar

As discussed previously, window aggregates are defined using the CREATE WINDOW AGGREGATE clause. The BNF grammar for window aggregate invocation is given in Figure 1. The grammar conforms to the SQL:2003 standards and adds the SLIDE construct discussed above.

Figure 2 summarizes how the powerful assortment of window types made available by ESL are reduced to the base and the (optional) window versions of each aggregate, using the techniques discussed in the previous section. Thus, Box (1) of Figure 2 represents the case of unbounded-preceding window, where we use the base aggregate and execute the TERMINATE state after processing each tuple. Box (2) shows the case where the simple window aggregate is called. Boxes (3) and (4) use the base aggregate to support tumbles, by restarting the computation after each new tumble, as discussed in Section 5.3. In addition, box (4) uses pane-based optimizations for an algebraic window aggregate (by cascading its tumble version with its window version), whereas box (5) describes the case when the aggregate is not algebraic, thus only

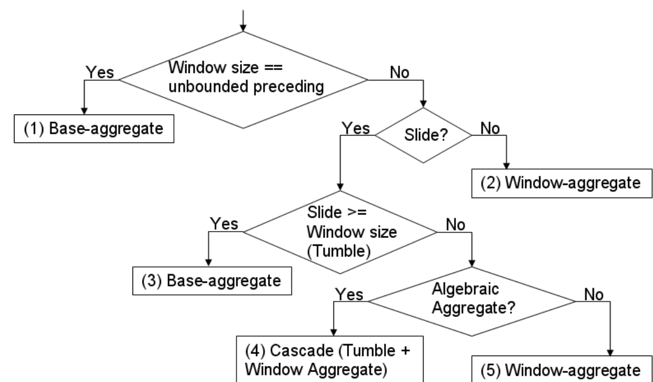

Figure 2: Evaluation of Window Aggregates

delta-maintenance is applied. Furthermore, by writing an explicit window version of the aggregate the user can achieve better performance and/or modify the basic behavior of the aggregate. However, if the user has not provided a window version of the aggregate then the system uses the default version as discussed in Section 4.

5.4 Evaluation of UDAs

Stream Mill translates ESL UDAs to C/C++ routines – one routine for each state, INITIALIZE, ITERATE, and so on. Upon arrival of a new input tuple, a base aggregate is executed as follows:

- Determine the group that the tuple belongs to.
- If this is the first tuple of the group then allocate an aggregate structure for the group and call the INITIALIZE routine of the aggregate. Store the newly allocated aggregate structure in a hash table with the group value as the key.
- If this is not the first tuple of the group, retrieve the aggregate structure from the hash table and call the ITERATE routine of the aggregate on the new tuple.

The above execution model is only slightly modified in the case of window aggregates invoked without the ACTIVE EXPIRATION clause. Upon arrival of a new tuple, the system first determines if there are any outstanding expired tuples. If outstanding expired tuples are found, their respective EXPIRE routines are invoked first. These expired tuples are subsequently removed from the **inwindow** buffer. Next, the newly arrived tuple is inserted into the **inwindow** buffer. Then, the system executes the ITERATE routine on the new tuple.

The execution of a window UDA invoked with the ACTIVE EXPIRATION clause takes place as follows: (i) the expiration of a tuple causes its EXPIRE statement to be executed, and (ii) the arrival of a new tuple only causes the ITERATE statement to be executed. Stream Mill employs a unique approach to enable the execution of the expire state exactly when the tuple expires. For each tuple in the original stream, a negative tuple with timestamp corresponding to the expiration timestamp of the original tuple is created. These negative tuples are then unioned with the original stream. The sort-merge semantics of the union statement delivers these tuples at exact times. When a negative tuple is seen in the resulting stream only the expire routine is invoked and the respective original tuple is removed from the **inwindow** table. A special delaying mechanism is used when the original stream has latent timestamps, since the union operator does not behave as a sort-merge in case of latent streams. More details of the union operator and different kind of timestamps can be found in [7].

The fact that the EXPIRE statements are written to support their active invocation when a tuple expires, allows us to 'garbage collect' expired tuples eagerly without waiting for new input tuples to arrive. This is particularly useful for time-based windows where all tuples are listed in their arrival order, and when a new tuple arrives all the expired tuples are processed, independent of their GROUP BY values.

This active expiration of windows is a source of great power and flexibility. For instance, it allows writing UDAs that detect that no tuple has arrived in the last 5 minutes (i.e., the expiring tuple was the last left in a window of 5 minutes) and send a warning message to the user. Moreover, they make it possible to support the update-pattern-aware processing of data stream via the "negative tuples" approach [22]. Remarkably, these advanced window types only require extensions to ESL UDA evaluation, and therefore do not affect the basic window types of SQL:2003.

Note that the execution model and the optimization strategies discussed here are also applicable to UDAs written in external language. The system calls external routines as opposed to the once generated by the ESL compiler. Pointers to the **inwindow** buffer, the return table, and any other user defined tables are passed to the external function. The external functions can access/manipulate these tables using the BerkeleyDB API [28]. Furthermore, external functions can also call special system functions such as the **oldest()**. The system implementer can take advantage of this flexibility to implement efficient aggregates in an external language.

5.5 Stream Mill Architecture

Stream Mill is implemented as a client-server architecture where several clients communicate with a single server. The **client** is used to manage streams, queries, aggregates, etc., and to view results of the user queries. The main components of the server are shown in Figure 3.

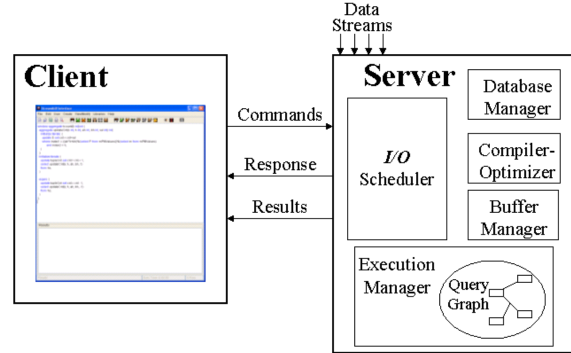

Figure 3: Stream Mill System Architecture

I/O scheduler is responsible for communicating with the clients and the **data manager**, which enables static data storage in Stream Mill.

Compiler/Optimizer is responsible for parsing and compiling of ESL queries. These queries are translated to C/C++ functions that employ numerous optimization techniques. Similarly, user-defined aggregates are also compiled to C/C++ functions. After careful optimizations, natively defined UDAs on the average execute nearly as well (typically a (10-20)% slowdown) as equivalent UDAs externally defined in a programming language (as shown in Section 6).

Buffer Manager is responsible for managing the stream tuples as in-memory queues. The Buffer Manager contains a special component called *Window Manager*, which is responsible for managing the **inwindow** buffers, including processing and expiration of tuples.

Execution Manager maintains a query graph containing all active queries in the system and makes operator scheduling decisions. More discussions on execution management and timestamp management can be found in [7].

6. EXPERIMENTS AND PERFORMANCE

We next describe the results of experiments that focus on testing the performance of new features of the Stream Mill System, such as window constructs for delta-computation and the pane optimization for slides and tumbles. In all the experiments the Stream Mill DSMS server was hosted on a Linux machine with P4 2.8GHz processor and 1GB of main memory.

6.1 New Constructs for Delta-Computation

In order to quantify the performance gain of the new window constructs for delta-computation and to study the performance of UDAs coded in ESL compared to a procedural language, we compared a few alternative implementations for a number of UDAs. We show graphs for two of these UDAs here, simple SQL-2 `average` and approximate frequent-items [13]. The frequent items algorithm

is a bloom filter based algorithm with two distinctions. First, there are k different hash-tables used, one for each hash function. Second, instead of storing a bit as an entry in the hash-table, it stores an integer to allow counting. For each incoming item, the algorithm looks up the hash entry in each of the k hash-tables and increments the counts. Similarly, for each expiring item the algorithm decrements the counts. Eventually, the approximate frequency of an item is given by the minimum value in the k hash-tables for that item.

Figure 4 and 5 compare the performance of the `average` and the approximate frequent-items aggregates, respectively, on different window sizes with the following implementations:

A) UDA in ESL using EXPIRE for delta computation

B) UDA in C using EXPIRE-like mechanism for delta computation (exactly the same operations as in A except it is coded in a procedure language)

C) UDA in ESL, performing delta computation without EXPIRE

D) Naive UDA in ESL (i.e., recompute for each new input tuple)

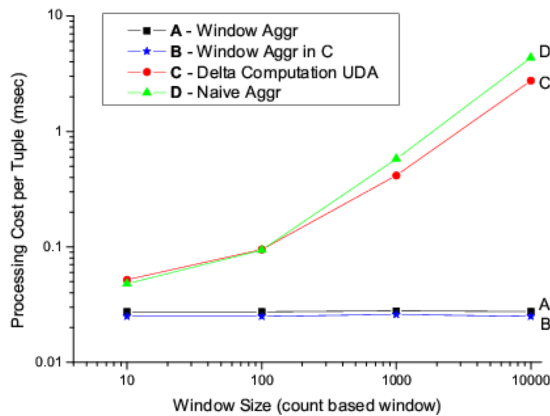

Figure 4: Average UDAs

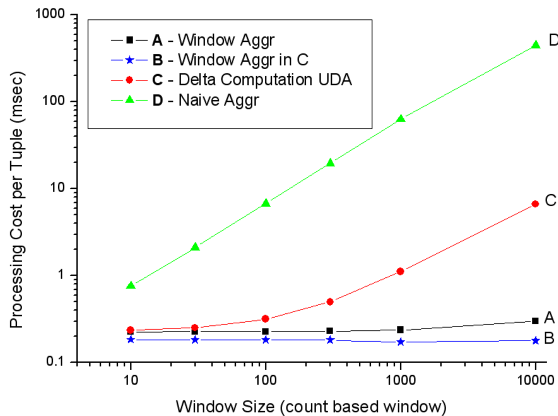

Figure 5: Approximate Frequency Count UDAs

There are two main conclusions that can be drawn from the above curves. First, the performance of the window aggregate with delta-optimization remains nearly constant independent of the size of the window. This is true whether the UDA is coded in ESL or in the C language, although the former is about (10-20)% slower. Thus, the overhead of UDAs coded in a high-level declarative language is minor. Secondly, the cost of the naive implementation, where the aggregate is recomputed on the whole window, grows linearly with the size of the window. The UDA expressing the delta computation without the support of the EXPIRE construct, (i.e. UDA version C above), also executes slower due to the fact that it has to pay the penalty of maintaining its own sliding window[7], thus making a clear case for the need of the new window constructs in terms of performance. The need in terms of user-convenience is even stronger, since without a language-supported EXPIRE construct, users will have to write different UDAs for logical and physical windows.

6.2 Slide and Tumble Based Optimization

As discussed in Section 5, Stream Mill takes the basic delta-based optimization for window UDAs and, for slides and tumbles, further optimizes it by exploiting the base version of the same UDA. To test the benefits so derived, we tested the processing speed and memory utilization of a sum UDA defined in ESL, on a window of 40000 tuples, for slides sizes ranging from 1 to the full window size.

SELECT **sum(price)** OVER (ROWS 40000 PRECEDING SLIDE N)
FROM **Bid**

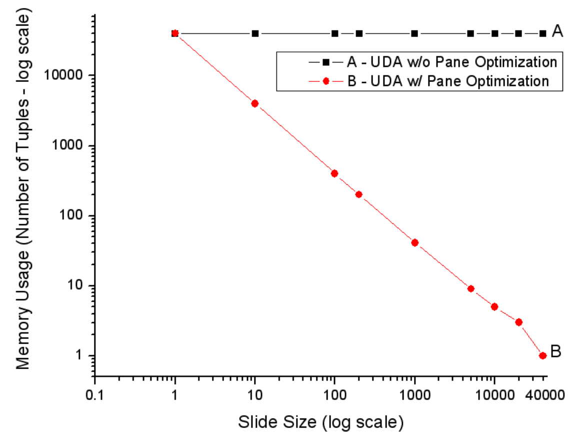

Figure 6: Memory Utilization for the Sum UDA

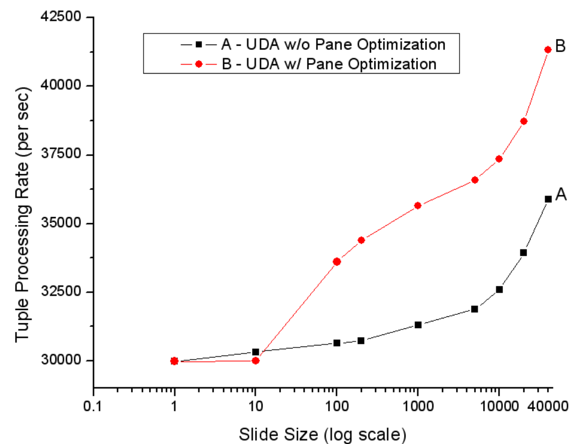

Figure 7: Tuple Processing Rate for the Sum UDA

We measure the memory usage and the tuple processing rate as shown in figures 6 and 7, respectively. The figures show two versions of the sum aggregate, one that takes advantage of the pane optimization along with delta-computation and the other that simply performs delta-computation. The optimization linearly reduces the

[7]For simple aggregates, such as avg, window maintenance can be as expensive as complete recomputation, as shown in Figure 4. This is no longer the case for more complex aggregates, such as that in Figure 5, where the cost of window maintenance does not dominate that of recomputation.

memory utilization as the slide size increases. The non-optimized version requires memory proportional to the window size, since tuples in the window are required to perform the delta-computation regardless of the slide value. However, the optimized version only keeps the aggregate value for each old pane in the window (plus the memory used for the computation of the aggregate on the current pane). For instance, if the window size is 40000 and the slide size is 10000, the system will partition the window in 4 panes of 10000 tuples each: thus it only needs to remember 4+1=5 values, instead of 40000. As shown in Figure 7, processing overhead is also reduced since the delta maintenance is performed every expiring pane as opposed to every tuple. Thus, the optimized version has a higher tuple processing rate, and this gain increases with increasing slide size—Figure 7. Thus, the slide and tumble optimization for algebraic UDAs improves both memory utilization and processing time. To the best of our knowledge StreamMill is the first DSMS that supports this optimization on arbitrary algebraic UDAs such as variance, string concatenation, other statistical functions, and some mining functions [25].

7. APPLICATIONS

UDAs have proven effective on a wide range of complex applications that include (i) Sampling and load shedding [20], (ii) Mining Data Streams [25], and (iii) Streaming XML queries [33].

The processing of sequences and time-series represent an important application area and there has been a significant amount of previous work trying to overcome the limitations of SQL in this area [27, 26]. However, we need to support continuous sequence queries on data streams since they can naturally be viewed as time-series. The effectiveness of UDAs in this role has been shown in [6].

Another important application of UDAs is in supporting approximate computation and synopses. For instance, the approximate evaluation of frequent items, using the algorithm proposed in [13], was recently discussed in [25]. Recently ESL was also used to implement the approximate computation example by Datar et al. [12] for 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.

Because of space limitations, we omit further discussions of these examples, and concentrate on the new and exciting area of RFID Data Processing.

Processing of RFID Data. RFID (Radio Frequency IDentification) technology is being deployed in a wide range of applications, including supply chain management, asset management, security, health care, etc. Frequently, raw RFID tag readings, which form continuous data streams, have to be processed to detect interesting temporal events, before further actions can be taken. While this problem can be addressed by ECA (Event Condition Action) rule systems, such as the deductive-rule based approach in [29], ESL proves to be a powerful and concise language for such temporal event detection.

Take as example a warehouse monitoring application, where RFID readers are used to automatically detect the presence of products, packaging boxes, personnel, etc. First consider a simple example of product-grouping detection.

EXAMPLE 17. *Suppose a group of products with RFIDs, which are being packed into the same case, are detected immediately following each other – i.e. the gap of detection time between consecutive products is below a certain time threshold t_1 seconds (e.g. 2 seconds). We like to detect this grouping of products automatically, and separate products into different groups if the gap of their arrival is longer than the threshold.*

While this example would be difficult to express in SQL, it is quite simple to write a UDA that compares the timestamp of newly arrived tuple with that of the last tuple stored in a local table. When the difference between the two timestamps exceeds the maximum allowed gap, then we increase the group number, otherwise we leave it unchanged. Finally, the input tuple is returned with a new column denoting the group number to which the item belongs.

For a more challenging 'real-time' application, consider the situation where RFID readers at the door detect products and personnel passing through. Say that only authorized personnel may carry products out of the door, either pushing or pulling a cart. We need to generate an alert when unauthorized people take out any product, which can be done as follows:

EXAMPLE 18. *If a product is detected at the door and there is no authorized personnel detected within time t_2 before or after the item exit, raise an alert of a potential theft.*

We can use the active expiration semantics to express this query. A window with time span equal to threshold t_2 is applied on the UDA. When an item expires from the window, we check that there is an authorized person (with code 'ap') event in the window, and send a probable-larceny alarm if condition fails. Moreover when an 'ap' event arrives, we remove all waiting 'item' tuples from the window. Suppose that the data stream schema is:

tag_readings(tagid, tagtype, tagtime); **theft**(itemid, etime);

Then we can use the following invocation where the window size is $t_2 = 5$ seconds (enough time for carts to move through the door).

```
CREATE STREAM larceny-in-progress AS (
SELECT tagid, tagtime, item_alarm(tagid, tagtype, tagtime) OVER
    (RANGE 5 SECONDS PRECEDING ACTIVE EXPIRATION)
FROM tag_readings);
```

The UDA can be expressed as follows:

```
AGGREGATE item_alarm(tagid, tagtype, tagtime):char
{    TABLE inwindow(tid, ttype, ttime);
     INITIALIZE: ITERATE: {
        DELETE FROM inwindow
           WHERE tagtype = 'ap' AND ttype = 'item';
        DELETE FROM inwindow
        WHERE tagtype = 'item' AND ttype = 'item' AND EXISTS
           (SELECT tid FROM inwindow WHERE ttype = 'ap'); }
     EXPIRE: {
        INSERT INTO RETURN VALUES ('larceny alarm')
           WHERE tagtype = 'item' AND NOT EXISTS
           (SELECT tid FROM inwindow WHERE ttype='ap'); }
}
```

8. RELATED WORK

Because of space limitations and the availability of authoritative surveys [4, 21] we will restrict our discussion to previous work on topics most significant to this paper, such as windows and UDAs. The importance of windows has been universally recognized in DSMS projects. For instance, the TelegraphCQ [19] project 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. The Aurora/Borealis projects provide extensive support for windows, including the concepts of slides and tumbles which have been adopted in ESL. The Gigascope project makes extensive use of windows to support network applications in network servers [10, 11].

Besides proposing techniques for an efficient window management [3], the STREAM project has used the notion of windows as a cornerstone for semantics of their Continuous Query Language CQL [2]. The view of window being an essential components of data streams has more recently inspired the 'update-pattern-aware' modeling of streams discussed in [22] and the temporal approach

proposed in [32]. Since the temporal and window-oriented semantics of streams is desirable in many applications [23], ESL has endeavored to provide support for this semantics using the 'active expiration' constructs discussed in Section 5. However, ESL retained the less specialized, and thus more general, 'append only' semantics, which is adopted by DSMS such as Aurora and Gigascope that have used it to support substantial real-life applications. These DSMSs make extensive use of UDAs (with windows) in their applications and Gigascope introduces very interesting aggregate constructs for sampling [20], however they did not explore the best language constructs for supporting all possible combinations of UDAs and windows, and the optimization that goes with them. Also, compatibility with SQL:2003 standards is not a stated objective for Aurora which instead relies on a "boxes and arrows" graphical interface to apply continuous queries and externally defined aggregates on data streams [14, 15].

Optimization issues for aggregates with windows have received much attention in the literature [3, 17, 9]. Thus the optimization techniques proposed for windows with panes in [17] have now been fully supported in Stream Mill using SQL:2003-based constructs that apply uniformly to built-in and user-defined aggregates, including very complex non-algebraic UDAs.

9. CONCLUSION

A key contribution of ESL is showing that there is no 'glass ceiling' that keeps a database-oriented approach from producing a general-purpose DSMS, while preserving SQL syntax and semantics. Indeed ESL has demonstrated its effectiveness on a broad range of applications that range from XML streams, to data mining and RFID-data processing, while minimizing extensions with current SQL:2003 standards. Compatibility with SQL standards is highly desirable for database vendors and database researchers alike, and will simplify applications that span both DB tables and data streams.

The reason for such superior power and flexibility is ESL's support of user-defined aggregate functions, that can be written in an procedural language or natively in ESL. In this paper, we have shown how to harness the great expressive power of UDAs [24] for data stream applications, by introducing a simple framework that integrates the various window constructs, and related optimizations, that were previously introduced in an ad-hoc fashion for built-in aggregates. The paper also describes the performance experiments that confirm the effectiveness of the proposed optimizations and the low overhead required to support powerful window UDAs.

The paper has also clarified the semantics of aggregates over different kinds of windows. We have also provided a simple union-based implementation for active-expiration semantics, which is important in real-time applications, and in achieving the update-aware semantics for data streams, recently advocated by some researchers [22].

10. REFERENCES

[1] Oracle http://www.oracle.com/technology/documentation/database10gr2.html.

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

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

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

[5] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *SODA*, 2002.

[6] Y. Bai, C. Luo, H. Thakkar, and C. Zaniolo. Efficient support for time series queries in data stream management systems. In *Stream Data Management–Chapter 6*. N. Chaudhry, K. Shaw and M. Abdelguerfi (EDs.), Kluwer, 2004.

[7] Y. Bai, H. Thakkar, H. Wang, and C. Zaniolo. Timestamp management and operator scheduling in Stream Mill DSMS. Technical report, Department of Computer Science, UCLA, June 2006.

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

[9] G. Cormode, F. Korn, S. Muthukrishnan, T. Johnson, O. Spatscheck, and D. Srivastava. Holistic UDAFs at streaming speeds. In *SIGMOD*, 2004.

[10] 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.

[11] C. Cranor, T. Johnson, O. Spatscheck, V. Shkapenyuk, and O. Spatscheck. Gigascope: A stream database for network applications. In *SIGMOD*, pages 647–651. ACM Press, 2003.

[12] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. In *SODA*, pages 635–644, 2002.

[13] C. Jin et al. Dynamically maintaining frequent items over a data stream. In *CIKM*, 2003.

[14] D. Abadi et al. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2):120–139, 2003.

[15] D. Abadi et al. The design of the borealis stream processing engine. *CIDR*, 12(2):120–139, 2005.

[16] D. Carney et al. Monitoring streams - a new class of data management applications. In *VLDB*, Hong Kong, China, 2002.

[17] J. Li et al. No pane, no gain: Efficient evaluation of sliding window aggregates over data streams. In *SIGMOD*, 2005.

[18] Jim Gray et al. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *J. Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

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

[20] Theodore Johnson et al. Sampling algorithms in a stream operator. In *SIGMOD*, pages 1–12, 2005.

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

[22] L. Golab and M. T. Özsu. Update-pattern-aware modeling and processing of continuous queries. In *SIGMOD*, pages 658–669, 2005.

[23] M. Hammad, W. Aref, M. Franklin, M. Mokbel, and A.K. Elmagarmid. Efficient execution of slidingwindow queries over data streams. Technical report, Department of Computer Sciences, Purdue University, December 2003.

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

[25] C. Luo, H. Thakkar, H. Wang, and C. Zaniolo. A native extension of SQL for mining data streams. In *SIGMOD*, pages 873–875, 2005.

[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] P. Seshadri, M. Livny, and R. Ramakrishnan. The design and implementation of a sequence database system. In *VLDB*, 1996.

[28] Sleepycat Software, http://www.sleepycat.com. *The Berkeley Database (Berkeley DB)*.

[29] Fusheng Wang, Shaorong Liu, Peiya Liu, and Yijian Bai. Bridging physical and virtual worlds: Complex event processing for RFID data streams. In *EDBT*, 2006.

[30] 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.

[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] Donghui Zhang, Dimitrios Gunopulos, Vassilis J. Tsotras, and Bernhard Seeger. Temporal aggregation over data streams using multiple granularities. In *EDBT*, 2002.

[33] X. Zhou, H. Thakkar, and C. Zaniolo. Unifying the processing of XML streams and relational data streams. In *ICDE*, 2006.