

A Native Extension of SQL for Mining Data Streams

Chang Luo[†]

Hetal Thakkar[†]

Haixun Wang[‡]

Carlo Zaniolo[†]

[†]Computer Science Department, UCLA, {lc,hthakkar,zaniolo}@cs.ucla.edu

[‡]IBM T.J. Watson Research Center, haixun@us.ibm.com

1 Introduction

ESL¹ enables users to develop stream applications in an SQL-like high level language that provides the ease-of-use of a declarative language, which is Turing complete in terms of expressive power [11].

In the database community, there is much interest in data stream mining applications [13, 16, 20] and in Data Stream Management Systems (DSMS) [21, 14, 19, 18]; but, while highly desirable, a marriage between the two faces difficult research challenges. The difficulty of the task is demonstrated by traditional databases where, due to query language limitations, mining tasks are typically carried out using a cache mining approach: i.e., by first moving the data from the DBMSs into a (memory or file based) cache, and then writing procedural programs to mine the cache [17]. But a similar solution is not possible for DSMS, since programming languages cannot handle well massive data streams unless we introduce primitives for managing buffers and windows, load shedding, query sharing and indexing, and other functions that duplicate those of DSMS [19, 18]. In this demo, we will show that this duplication can be avoided by using ESL, which extends SQL-based continuous query systems by enabling them to support stream mining effectively and efficiently.

The latest version of ESL is available for download [1] along with a suite of applications, which include stream data mining functions, that have been coded in ESL, and execute with a modest performance overhead with respect to the same applications written in C/C++. Our SIGMOD 2005 demo will illustrate the key features and applications of ESL. In particular, we will demonstrate how to:

- declare and access streams in ESL;
- create new user-defined aggregates and functions in ESL;
- apply aggregates on streams using count-based or time-based windows;
- code concisely complex data mining applications in ESL for data streams, including for example, approximate stream queries [12], and classifiers maintained over concept-drifting data streams
- Various GUI tools that can be used to visualize continuously the results of our mining algorithms, their performance, and the performance of the ESL system.

¹ESL stands for: Expressive Stream Language

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.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

2 A Stream Language based on SQL

There is much current research on stream queries [19, 10, 3]. ESL, through the introduction of a minimal set of new language constructs, supports advanced applications that span both data streams and stored database tables.

Stream Declaration ESL views data streams as unbounded ordered sequences of tuples, which is consistent with the ‘append only table’ model commonly used by data stream systems. In ESL, each data stream is imported from an external wrapper via the (mandatory) SOURCE clause in its CREATE STREAM declaration. This declaration also specifies the timestamp associated with the stream. For instance, in Example 1, the data stream OpenAuction is declared as having **start.time** as its external timestamp.

EXAMPLE 1. *Declaring Streams in ESL*

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

In ESL, new streams can be defined from existing streams in a way similar to defining 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
```

User Defined Aggregates (UDAs) UDAs are important in advanced database applications including decision support and stream queries [7, 19, 4]. ESL adopts from SQL-3 [5] 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 three procedures coded in procedural languages as in SQL-3². This approach makes ESL expressive and extensible [11].

Example 3 defines an aggregate equivalent to the standard **avg** aggregate in SQL. The first line of this aggregate function declares a local table, **state**, to keep the sum and count of the values processed so far. The remaining SQL statements are grouped into the three blocks labelled respectively INITIALIZE, ITERATE, and TERMINATE. Thus, the INITIALIZE statement inserts the value taken from the input stream and sets the count to 1. The ITERATE statement updates the table by adding the new input value to the sum and 1 to the count. The TERMINATE statement returns the ratio between the sum and the count as the final result of computation. This is done by INSERT INTO RETURN statement that in this case only returns one value but, in general, could return several values.

²Although UDAs have been left out of SQL 1999 specifications, they were part of early SQL-3 proposals, and supported by some commercial DBMS.

EXAMPLE 3. *Defining the standard aggregate average*

```

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

```

Observe that each group of SQL statements in INITIALIZE, ITERATE, and TERMINATE plays the same role as the external functions in SQL-3 aggregates. But here, we have assembled the three functions under one procedure, thus supporting the declaration of their shared tables (the *state* table in this example).

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

In Example 4, a physical window of 10 items (ROWS 9 PRECEDING) is maintained for each seller (PARTITION BY *sellerID*) to compute the average price of the 10 items in the window.

EXAMPLE 4. *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)
FROM ClosedPrice;

```

Example 5 defines an optimized implementation of *avg*. 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 UDA. For *avg*, the delta maintenance consists in decreasing the count by one and the sum by the value of the expired tuple—*oldest()* returns the value of the oldest of tuple in the window buffer. Also, EXPIRE is executed once for each expiring tuple.

The result is the same whether this delta computation is performed as soon as the 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.

EXAMPLE 5. *Defining avg on Sliding Windows*

```

WINDOW AGGREGATE avg(Value REAL) : REAL
{
  TABLE state(tsum INT, cnt REAL);
  INITIALIZE : {
    /* here, Value is the new tuple coming into the window */
    INSERT INTO state VALUES (Value, 1);
  }
  ITERATE : {
    /* here, Value is the new tuple coming into the window */
    UPDATE state SET tsum=tsum+Value, cnt=cnt+1;
    INSERT INTO RETURN
    SELECT tsum/cnt FROM state;
  }
  EXPIRE : {
    /* here, Value is the tuple expired from the window */
    UPDATE state
    SET cnt= cnt-1, tsum = tsum - oldest();
  }
}

```

Example 3 and Example 5 illustrate the language constructs introduced by ESL to express new UDAs. These two are extremely

simple UDAs, however, the language constructs are powerful enough to express many advanced database and data stream applications. Complex applications, including decision tree classifiers, association rule mining, and other data mining functions can be concisely written and efficiently implemented in ESL [6, 7, 8]. ESL also makes available to users a wide variety of approximate, and application-specific aggregates needed for stream applications.

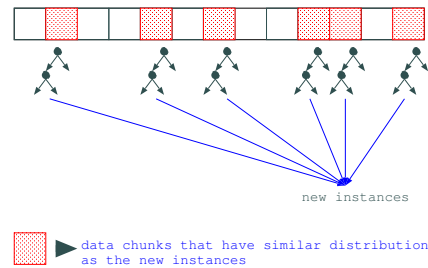
3 Mining Data Streams with Concept Drifts

ESL is capable of expressing many advanced applications, including those that span databases and data streams. In our demo, we show this by using ESL to implement one of such applications: mining data streams with time-changing concept drifts.

Streaming data is characterized by its time-changing concepts as well as its huge data volume. In other words, the model that we try to learn from the data is constantly evolving. Thus, a fundamental problem we need to solve is the following: given an infinite amount of continuous measurements, how do we model them in order to capture time-evolving trends and patterns in the stream, and make time-critical predictions?

One approach is to incrementally maintain a model for the time-changing data. The model is learned from data in the most recent window. This approach has several weak points. First, given that data arrive at a high speed, incremental model maintenance is usually a costly task, especially for learning methods such as the decision tree algorithm, which is known to be unstable. Second, models trained from the data in a window may not be optimal. If the window is too large, it may contain concept drifts; if it is too small, it may result overfitting.

We use ESL to implement a more effective approach [13]. We partition a stream into fixed size data chunks and learn a model from each chunk. We combine models learned from data chunks whose class distribution is similar to the most recent training data to be our stream classifier (Figure 1). This approach reduces classification error in the concept-drifting environment.



1: Mining Streams with Concept-Drifts

The seemingly complex solution described above can be implemented in ESL in a very succinct way. We assume each example in the stream is in the form of (a_1, \dots, a_n, L) , where a_1, \dots, a_n are attribute values, and L is the class label. If $L = TBA$ then it is a testing example, otherwise it is a training example. In Example 6, we express the algorithm in one SQL statement.

Here, we call UDA *ClassifyStream* with keyword *SLIDE*, which implements data partitioning on the stream. We refer the reader to [15] for a detailed description of ESL's support of the *slide*, which is a useful extension of SQL:2003 for DSMs [14]. In the example below, the slide and the window have the same size (10000), which constitutes a special slide called the *tumble* [14]. For tumbles, the computation is re-started with each new partition and terminated at the end of the same partition—as opposed to the case where the window slides of one tuple and the old aggregate value is instead continuously revised (see Example 8).

We assume classifiers, together with their weights, are stored in a table called *ensemble*. In UDA *classifystream*, we use classifiers in ensemble whose weights are above a given threshold to classify each test example, where *Classify* is a UDA for classifying static

data [9]. Once we reach the end of a data partition, we learn a new classifier from the training data in the partition, and we reset the weight of each classifier in ensemble by its accuracy in classifying the training data. The freshly weighted classifiers will then be used to classify data in the next partition.

EXAMPLE 6. *A Terse Expression for Complex Classifier Ensembles*

```
SELECT ClassifyStream(S.*)
OVER (ROWS 10000 PRECEDING SLIDE 10000)
FROM stream AS S;
```

EXAMPLE 7. *UDA classifystream*

```
AGGREGATE ClassifyStream(a1, ..., an, L) : Int
{
  INITIALIZE : ITERATE : {
    INSERT INTO RETURN
      SELECT sum(E.Classify(a1, ..., an) × E.weight) /
             sum(E.weight)
      FROM ensemble AS E
      WHERE L = TBA AND E.weight ≥ threshold;
  }
  TERMINATE : {
    INSERT INTO ensemble
      SELECT learn(W.*) FROM WINDOW AS W
      WHERE W.L <> TBA;
    UPDATE ensemble AS E SET E.weight =
      (SELECT 1-avg((E.Classify(W.*)-W.L))
      FROM WINDOW AS W
      WHERE W.L <> TBA);
  }
}
```

4 Maintaining Frequent Itemsets over a Stream Window

Mining frequent itemset on static datasets has been studied extensively. However, data streams have posed new challenges. First, we want to avoid multiple scans of the data. Second, because the data distribution is usually changing with time, we are more interested in most recent patterns.

One approach is to always focus on frequent itemsets in the most recent window. We can i) regenerate frequent itemsets from the entire window, or ii) store every itemset, frequent or not, and update its support whenever the content of the window changes.

It is clear that neither method is efficient. Therefore, in our demo, we will use the new approach proposed in [16], which, instead of monitoring every itemset, uses a *dynamic set*. The dynamic set is i) informative enough to answer at any time “what are the frequent itemsets in the current window”, and ii) small enough so that they can be maintained easily and updated in real time. The dynamic set consists of frequent itemsets, and itemsets that form the boundary between the closed frequent itemsets and the rest of the itemsets [16]. It can be shown that i) a status change of any itemset (e.g., from non-frequent to frequent) must come through the boundary itemsets, which means we do not have to monitor itemsets beyond the boundary [16].

In our implementation, we maintain the dynamic set by three external C++ functions, *explore*, *add*, and *delete*, which creates an internal data structure for the dynamic set, increases the count of involved itemsets, and decreases the count of involved itemsets in the dynamic set, respectively (for algorithmic details of these external functions, see [16]). Example 8 shows how ESL interfaces with external functions to maintain frequent itemsets over streams.

EXAMPLE 8. *Maintaining Closed Frequent Itemsets over Streams*

```
WINDOW AGGREGATE cfi(t Set) : Int
{
  INITIALIZE : { explore(t); }
  ITERATE : { add(t); }
  EXPIRE : { delete(oldest()); }
}
SELECT cfi(s.transaction) OVER(ROWS 1000 PRECEDING)
FROM stream AS s;
```

Example 7 and Example 8 are extreme cases of using UDAs. In the first case, everything is done in ESL and in the second case, we call C++ function that manage user-defined data structures. The DSMS plays an important role in the second case as well, because it manages the windows and expiring tuples, and it is used for additional tasks, such as cleaning the data, deriving the rules, etc.

The most recent version of ESL, the Stream Mill system, and reports on current developments, can be downloaded from [1].

5 References

- [1] The Stream Mill Project. <http://wis.cs.ucla.edu/stream-mill/>
- [2] Brian Babcock, Shivnath Babu, Rajeev Motwani, Jennifer Widom. Models and Issues in Data Streams, *PODS*, 2002.
- [3] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. *SIGMOD*, pages 379-390, 2000.
- [4] P. Domingos and G. Hulten. Mining high-speed datastreams. *SIGKDD*, pages 71-80, 2000.
- [5] ISO/IEC JTC1/SC21 N10489, ISO/IEC 9075, “Committee Draft (CD), Database Language SQL”, July 1996.
- [6] Haixun Wang and Carlo Zaniolo. Using SQL to Build New Aggregates and Extenders for Object-Relational Systems. *VLDB*, 2000.
- [7] Haixun Wang, Carlo Zaniolo. Extending SQL for Decision Support Applications. *DMDW*, 2002.
- [8] Haixun Wang, Carlo Zaniolo. ESL: A Native Extension of SQL for Data Mining and Stream Computations *UCLA CS Dept, Technical Report*
- [9] Haixun Wang, Carlo Zaniolo. ATLAS: A Native Extension of SQL for Data Mining. *SIAM DM*, 2003.
- [10] Sam Madden, Mehul A. Shah, Joseph M. Hellerstein, Vijayshankar Raman. Continuously Adaptive Continuous Queries over Streams. *SIGMOD*, pages 49-61, 2002.
- [11] Yan-Nei Law, Haixun Wang, Carlo Zaniolo. Query Languages and Data Models for Database Sequences and Data Streams. *VLDB*, 2004.
- [12] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows: (extended abstract). In *SODA*, 2002.
- [13] Haixun Wang, Wei Fan, Philip S. Yu, and Jiawei Han. Mining Concept-Drifting Data Streams using Ensemble Classifiers. *SIGKDD*, 2003.
- [14] 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. *VLDB* 2002.
- [15] WEB Information System Laboratory UCLA. An Introduction to the Expressive Stream Language (ESL). <http://wis.cs.ucla.edu>.
- [16] Yun Chi, Haixun Wang, Philip Yu, and Richard Muntz. Moment: Maintaining Closed Frequent Itemsets over a Stream Sliding Window. *ICDM*, 2004.
- [17] S. Sarawagi, S. Thomas, R. Agrawal. Integrating Association Rule Mining with Relational Database Systems: Alternatives and Implications. *SIGMOD*, 1998.
- [18] Lukasz Golab and M. Tamer -zsu. Issues in data stream management. *ACM SIGMOD Record*, 32(2):5-14, 2003.
- [19] B. Babcock, S. Babu, M. Datar, R. Motawani, and J. Widom. Models and issues in data stream systems. *PODS*, 2002.
- [20] Fang Chu and Carlo Zaniolo. Fast and Light Boosting for Adaptive Mining of Data Streams. *PAKDD* 2004:
- [21] A. Arasu, S. Babu, and J. Widom. An abstract semantics and concrete language for continuous queries over streams and relations. Technical report, Stanford University, 2002.