ESL : a Data Stream Query Language and System Designed for Power and Extensibility

UCLA CSD Technical Report TR040030

Chang Luo
University of California, Los Angeles
lc@cs.ucla.edu

Haixun Wang
IBM T. J. Watson Research Center
haixun@us.ibm.com

Carlo Zaniolo
University of California, Los Angeles
zaniolo@cs.ucla.edu

Abstract

Applications that span both data streams and databases provide a sound reason for using SQL for continuous queries on data streams as well as for ad-hoc queries on database tables. However, SQL was designed for persistent data on secondary store, and not for transient data on the wire; therefore, SQL is facing major problems in this new role—particularly in the domains of expressive power and extensibility that constitute notorious weakness areas for relational languages. For instance, SQL cannot support sequence queries, approximate queries, and mining queries that represent important applications for data stream management systems (DSMS). The fact that blocking operators cannot be used on continuous queries further impairs the effectiveness of SQL on data streams applications. Our Stream Mill solves these problems with the Expressive Query Language (ESL) designed to maximize the power and flexibility of DSMS through minimal extensions of, and maximal compatibility with, SQL:2003. ESL supports (i) continuous user-defined aggregates (UDAs) that are nonblocking and very effective on data streams, (ii) efficient delta-based maintenance for UDAs on windows, (iii) table expressions to create table snapshots from data streams. With these constructs, we achieve full integration of database and stream queries while retaining a simple semantics which makes it crystal clear which queries run as snapshot queries and which run as continuous queries, and which constructs are allowed for each kind. The paper discusses the benefits of the power of ESL (which is Turing-complete) in advanced applications and its ability to support efficient memory-saving computations.

1. Introduction

There is much ongoing research work on data streams and continuous queries [2, 15]. The Tapestry project [5, 38] was the first to focus on the problem of 'queries that run continuously over a growing database'. Recent work in the Telegraph project [7, 25, 12] focuses on efficient support for continuous queries and the computation of standard SQL-2 aggregates that combine streams flowing from a network of nodes. The Tribeca system focuses on network traffic analysis [37] using operators adapted from relational algebra. The OpenCQ [23] and Niagara Systems [8] support continuous queries to monitor web sites and similar resources over the network, while the Chronicle data model uses append-only ordered sets of tuples (chronicles) that are basically data streams [20].

The Aurora project [6] aims at building data management systems that provide integrated support for

- Data streams applications, that continuously process the most current data on the state of the environment.
- Applications on stored data (as in traditional DBs).
- Spanning applications that combine and compare incoming live data with stored data.

The need to support spanning applications implies the desirability of using, insofar as possible, the same query language on data streams and stored tables. This is clearly the approach of choice since it maximizes the compatibility with existing information systems and greatly reduces the efforts required for learning and using DSMS. But designing a language for a new application domain represents a research challenge even in the most favorable of circumstances, and, in our case, the requirement of building on SQL, which was designed for a different application environment and enabling technology, brings additional problems. Indeed, SQL was designed for transient queries on persistent data, rather than for persistent queries on transient data flowing in on the wire.

The difficult challenge of designing an SQL-based language for DSMS was addressed by the Stanford Stream project with the design of the Continuous Query Language (CQL) [1]. CQL introduces several stream-oriented concepts and SQL-based constructs, for which a rigorous semantics is also given [1]. Windows represent a very important construct in CQL, insofar as they are used in the computation of aggregates and joins on streams, and to map from streams to relations. Although CQL addresses many of the requirements of simple data stream queries, it does not achieve the level of expressive power needed for more complex data stream applications, which are instead the focus of our ESL language discussed next.

The Stream Mill project [11] seeks to overcome the serious limitations of SQL on data stream applications, while minimizing the deviations from current standards, by supporting the Expressive Stream Language, ESL. Some SQL limitations had already surfaced in DBMSs applications, others are new and specific to data stream applications. Among the latter, we find that blocking query operators [2] cannot be used on data streams, whereas they provide the cornerstone of many SQL constructs and implementation technology. The SQL-2 constructs that cannot be used on data streams include all of the SQL-2 aggregates and the constructs NOT IN, NOT EXISTS, ALL, EXCEPT.

An in-depth analysis of the blocking problems and its impact on expressive power was recently presented in [22]. It was there proven that (i) a function can be expressed by a nonblocking query iff it is monotonic, and (ii) SQL is not relationally complete on data streams, since it can only express some of its monotonic queries by using blocking operators, which are disallowed on data streams. This loss of expressive power represents a serious setback for SQL, since its inadequacies in advanced application domains have long been lamented by database researchers. Among these problem areas we find time series queries [36, 35, 30, 28, 33, 32] and data mining queries [16, 26, 19, 34] on stored databases. These are major limitations, since mining data streams represent an area of growing interest, and sequence queries are very badly needed inasmuch as data streams are infinite time series. These traditional SQL problems appear to impact primarily complex queries, but its inability of expressing monotonic queries expressible in SQL affects simple data stream queries besides complex ones.

Furthermore, SQL problems are exacerbated by yet another difference with respect to traditional database applications. Complex database applications are normally written by embedding SQL queries in a procedural program via cursor-based interface mechanisms. Then, functions that are too complex for SQL can be written in the procedural language. In the cursor-based model of embedded queries the procedural program pulls tuples from the database via get-next statements. But for data streams that arrive continuously, at a fast and often bursty pace, it is unreasonable to expect that the DSMS can hold on to the current tuple—and those that have arrived after that—until the application issues its get-next statement. Indeed, most of current data stream management prototypes do not support cursor-based interfaces to programming languages.

Therefore, the first objective of ESL is to enhance the expressive power of SQL in ways that support well typical data stream applications, including those featuring synopses, memory minimization, approximate queries, sequence queries, and stream mining queries. ESL pursues this mission along with the two complementary objectives of achieving (i) minimal deviations from the latest SQL:2003 standard, and (ii) simple semantics that clearly separate continuous queries from ad hoc queries, and identify which constructs can be used for the kinds of queries.

The main language constructs of ESL are as follows:

- Data stream declarations are used to import streaming data (from external wrappers) and assign timestamps to them. Then, ESL queries consisting of nonblocking SQL operators are used to create new data streams to be supplied to other queries, or returned to the user. When applied to data streams, the union operator performs a sort-merge operation that combines its input streams according to their timestamps. These topics are discussed in the next section.
- User-defined aggregates (UDAs) natively defined in SQL provide the main mechanism to achieve expressive power and extensibility for ESL. UDAs come in two versions easily identified from their syntax: nonblocking UDAs can be used on both tables and streams, while blocking UDAs can only be used on tables. UDAs are introduced in Section 3, and in Section 4, we demonstrate their uses in memory management, approximate computations and sequence queries.
- Window constructs are presented in Section 5 where we focus on their efficient implementation and the incremental, delta-based, maintenance of UDAs on windows.
- ESL uses table functions to derive from data streams concrete table-like views that support ad-hoc queries. These are discussed in Section 6, where we also propose a simple semantics for queries spanning both data streams and database tables.

In summary, the paper is organized along the two parallel themes of introducing (i) the simple constructs of ESL, and (ii) the complex applications that can be efficiently supported with those constructs. The discussion is completed by Section 7, which focuses on complex applications, and Section 8, which discusses related work.

2. Streams and timestamps

ESL views data streams as unbounded ordered sequences of tuples [22]; this is consistent with the 'append only table' model commonly used by data stream systems [5, 2, 15, 22]. In ESL, data streams are imported from external wrappers by declarations that specify their source wrapper and the kind of timestamp used. While, Stream Mill wrappers[1] are very similar to those used in other systems [12, 1], timestamps are quite different. In ESL, during their declarations, data streams can be assigned *external timestamps*, *internal timestamps* or *latent timestamps*. External timestamps are produced by the application generating the data and correspond to a column of the tuple incoming from the the wrapper. Instead, *internal timestamp* are generated by the system as the tuples arrive from the buffer.

For instance, in the following on-line auction example, adapted from [3], **OpenAuction** and **Bid** streams have external timestamps while **ClosedAuction** has internal one.

```
/* Stream of auction openings */
CREATE STREAM OpenAuction (
                itemID /* id of the item being auctioned.*/,
                sellerID /* seller of the item being auctioned.*/,
                start_price /* starting price of the item */,
                timestamp /* time when the auction started */)
ORDER BY timestamp; /* external timestamp */
SOURCE 'OpenAuction.so';

/*Stream of auction closings */
CREATE STREAM ClosedAuction(
                 itemID /* id of the item in this auction. */,
                 buyerID /* buyer of this item.*/)
ORDER BY INTERNAL; /* internal timestamp */

/* Bid: Stream of bidding. */
CREATE STREAM Bid(
      itemID /* the item being bid for*/,
      bid_price, /* bid price */,
      bidderID /* id of the bidder*/,
      timestamp /* time when bid was registered */)
ORDER BY timestamp; /* external timestamp*/
```

While the SOURCE clause specifying the external buffer cannot be omitted from the declarations, the ORDER BY clause can be omitted altogether. This is the case of *latent timestamps*—as opposed to the external timestamps and internal timestamps which will call *explicit*. Latent timestamps are not kept in the actual tuples, and cannot be used in the queries, but are expediently generated by system when these tuples are accessed. Latent timestamps can produce better performance, and the convenience of operational semantics when needed.

For instance, for externally timestamped streams, there is no assurance that their tuples arrive exactly sorted according to their timestamps. ESL solves this problem by reassigning all late tuples to a separate stream with a latent timestamp, which can then be handled directly to the user.

1 Basically, data source can be programmed using external procedural language such as C and compiled into a dynamic library.

2.1. UNION

Union is the only n-ary operator ($n > 1$) supported by ESL on data streams. For instance, the following query sort-merges two streams according to their explicit timestamps into an explicitly timestamped **history1000** query:

Example 1 *Price History Query: Report price history for item #1000*

```
CREATE STREAM history1000 AS
      SELECT itemID, start_price, timestamp
      FROM OpenAuction WHERE itemID = 1000
      UNION ALL
      SELECT itemID, bid_price, timestamp
      FROM Bid 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 produced an explicitly timestamped data stream; likewise, the union of two or more data streams with latent timestamp produces a data stream with latent timestamps.

In implementing the union of streams with latent timestamps, the system goes round-robin to take from the buffers the tuples currently there; while the system is designed to service all input buffers fairly, the output order of the tuples only preserves the order of each input stream.

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.

3. A Small but Complete Extension of SQL

User Defined Aggregates (UDAs) are important for decision support, stream queries, and other advanced database applications [41, 2, 17]. 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 [40]— rather than by three procedures coded in procedural languages as prescribed by SQL-3[2]. Readers who are already familiar with UDAs can skip to the next section.

Example 2 defines an aggregate equivalent to the standard AVG aggregate in SQL. The second line in Example 2 declares a local table, **state**, where the sum and count of the values processed so far are kept. Furthermore, while in this particular example, **state** contains only one tuple, it is in fact a table that can be queried and updated using SQL statements and can contain any number of tuples. These SQL statements are grouped into the three blocks labeled, respectively, INITIALIZE, ITERATE, and TERMINATE. Thus, INITIALIZE inserts the value taken from the input stream and sets the count to 1. The ITERATE statement updates the tuple in **state** by adding the new input value to the sum and 1 to the count. The TERMINATE statement returns the ratio between the sum and the count as the final result of the computation by the INSERT INTO RETURN statement[3]. Thus, the TERMINATE statements are processed just after all the input tuples have been exhausted.

Example 2 *Defining the standard aggregate average*

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

Observe that the SQL statements in the INITIALIZE, ITERATE, and TERMINATE blocks play 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). This table is allocated just before the INITIALIZE statement is executed and deallocated just after the TERMINATE statement is completed. This approach to aggregate definition is very general. For instance, say that we want to support tumbling windows of 200 tuples [6]. Then we can write the UDA where the RETURN statements appear in ITERATE instead of TERMINATE.

In Example 3, we define a **maxpair** aggregate that returns the maximum value along with the first point at which the maximum occurred. Observe that the formal parameters of the UDA function are treated as constants in the SQL statements. Thus, the UPDATE statement in ITERATE store the constant **iPoint** in the **mpair** relation, provided that **iValue** is larger than the value in **mpair**. Therefore, the WHERE clauses operate here as conditionals. The RETURN statement in TERMINATE returns the maximum value and the first point where it occurred.

Example 3 *Define* **maxpair** *in ESL*[4]

```
AGGREGATE maxpair(iValue, iPoint):(mValue, mPoint)
{     TABLE mpair(value, point);
      INITIALIZE:{
         INSERT INTO mpair VALUES(iValue, iPoint);
      }
      ITERATE:{
         UPDATE mpair SET point=iPoint, value=iValue
               WHERE iValue > value;
      }
      TERMINATE: {
         INSERT INTO RETURN
            SELECT value, point FROM mpair
      }
}
```

Observe that in SQL, the expression of this query is somewhat contrived and inefficiency prone (these problems escalate rapidly as we move from simple queries to more complex ones). In fact, the computation will have to be expressed as follows: the MAX aggregate will first be used to find the maximum value; then, this can be used as a subquery of another query that finds all points where this maximum occurs and return the first occurrence of these points. It can be shown that UDA **maxpair** takes linear time to the size of the original table while the equivalent SQL query takes quadratic time.

Blocking versus Non-Blocking UDAs The UDAs in the previous examples are blocking [22] since they return values in their TERMINATE state. Blocking UDAs cannot be used on data streams. On the contrary, UDAs whose TERMINATE state is either empty or missing can only return values in their ITERATE or INTITIALIZE states, and are therefore nonblocking [22]. Non-blocking UDAs can be freely used to express continuous queries on data streams [5]

For instance, we can build a continuous version of the average aggregate of Example 2, by simply moving the statements in TERMINATE to ITERATE. The new UDA so constructed is nonblocking and can be used freely on data streams.

2 Although UDAs have been left out of SQL:2003 specifi cations, they were part of early SQL-3 proposals, and supported by some commercial DBMS.

3 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.

4 We omit data type defi nitions in the following examples

5 Moreover, as we shall see later, nonblocking UDAs that use auxiliary tables (including windows) can apply blocking computation on these tables without losing their non-blocking properties.

4. UDA on Data Streams

The power of UDAs has been studied in [22] where we have shown that turn SQL into a Turing complete language that can express every query computable from the database. We have also shown that non-blocking UDAs are also complete on data streams, inasmuch as they can be used to express every computable monotonic query. We next show that such non-blocking UDAs are indeed very effective in dealing with data stream applications.

4.1. 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 5). 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 4 *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;
      }
}
```

4.2. Joins by UNION and UDAs

UNION operations are very useful in many applications. For instance, to compute the closing price of an item (the maximum bid price, or its starting price if no bids has been placed), a naive approach will involve a join of 3 streams, which requires unbounded memory. A more efficient implementation is shown in Example 5, below, where we union 3 streams together: **OpenAuction**, **Bid** and **ClosedAuction** into a new stream, **AllPrice**, which is sorted by the arrival timestamp. Tuples in **AllPrice** are tagged by **start**, **bid**, and **end**, which indicate their origin. Once the three streams are unioned together, we can call UDA **mymax** that memorizes the maximum price seen so far and returns this maximum when an **end** tag is seen. This query returns its results directly to stdout.

Example 5 *Closing Price Query: Report the closing price of each auctioned item.*

```
INSERT INTO stdout
SELECT itemID, mymax(price, tag) AS Price
FROM
      ( SELECT itemID, start_price, timestamp, 'start'
      FROM OpenAuction
      UNION ALL
      SELECT itemID, bid_price, timestamp, 'bid'
      FROM Bid
      UNION ALL
      SELECT itemID, -1, timestamp, 'end'
      FROM ClosedAuction /* ClosedAuction has no price */
) AS   AllPrice (itemID, price, timestamp, tag)
GROUP BY itemID;
```

In the above example, the result of the UNION is a data stream. UDA **mymax** is grouped by **itemID** so that it applies to different items separately. UDA **mymax** memorizes the maximum price seen so far and returns this maximum when an end tag is seen. This UDA is shown below:

```
AGGREGATE mymax(price, tag) : (price)
{     TABLE memo(mprice) MEMORY;
      INITIALIZE:{
         INSERT INTO memo VALUES(price);
      }
      ITERATE:{
         UPDATE memo SET mprice = price
            WHERE price > mprice;
         INSERT INTO RETURN
            SELECT * FROM memo WHERE tag = 'end';
      }
}
```

Using union and UDAs, we have here avoided having to join the three streams. Here, for each open item, the in-memory table memo contains exactly one tuple—the maximum. Therefore this approach requires memory linear in the number of open items—rather than the unbounded memory that might be required for joins.

4.3. Approximate Computations

Approximate computations and synopses are very important on data streams and can be expressed very effectively in ESL. While windows are very important, many approximate computations for data streams are not based on windows. The computation of decaying averages, where e.g., new incoming tuples are assigned a slightly greater weight than the current average— can be expressed by a simple modification of our Example 2.

For a more interesting example, consider the on-line synopsis algorithm by Datar et al. [10]. The problem is the following: given a stream of data elements consisting of 0's and 1's, maintain at every time instant an approximate count of the number of 1's in the last N elements. Datar's algorithm assures a relative error bound of $1 + \epsilon$ and uses $\Omega(\frac{1}{\epsilon} \log^2 N)$ bits of memory space. The algorithm is as follows.

If an incoming element is 1, create a new bucket of value 1. If there are k neighboring buckets with same value v, merge the oldest two buckets into one bucket with value $2v$ (value k depends on the error bound ϵ, which is given by the user). At every time instant, the approximate count is the sum of the values of all buckets minus half of the value in the oldest bucket.

Example 6 *Approximate Counts over Streams*

```
AGGREGATE basicCount(next Int, t Timestamp, k Int) : {
    TABLE hist(h Int, t Timestamp);
    INITIALIZE : ITERATE : {
      INSERT INTO hist VALUES(next, t) WHERE next = 1;
      DELETE FROM hist h WHERE h.t < t - T;
      INSERT INTO RETURN
        SELECT merge(h, t, k)
    }
}
```

Example 6 implements the above procedure in ESL. In the code, we use a table hist to maintain the buckets, and calls another UDA **merge** to find tuples of same value and merge them. Due to space limitations, we do not list here the code for the UDA **merge**, which is posted in [11].

5. Windows

The window construct is very important for data streams applications [1, 6]. Therefore, the decision was taken that ESL would support windows on arbitrary UDAs, although the problem is significantly more difficult than supporting windows on standard built-in aggregates. We addressed this problem with the introduction of constructs that support the delta-based maintenance of aggregate on windows.

In SQL:2003, windows are used as *aggregate modifiers* applied to ordered sequences of tuples. ESL extends this role to windows applied to data streams—indeed, data streams can be viewed as temporally ordered sequences of unbounded length. However, ESL preserves the syntactic framework of SQL:2003, whereby windows are used as aggregate modifiers which can be appended to the aggregate call via the OVER clause (whereas, in other DSMS windows have been added to the 'from' clause). This example illustrates how the design of ESL strives to maximize the expressive power and efficiency of the language while attempting to minimize the syntactic deviations from SQL standards.

Both logical (time-based) and physical (count-based) windows are supported. In the next example, we assume that we already have the **ClosedPrice** stream (see Example 5 for the actual computation of this stream). Then, a window with a capacity of 10 items (ROWS 9 PRECEDING) is maintained for each seller (PARTITION BY sellerID), and the average price of the 10 items in the window is returned.

Example 7 *Count-Based Window : For each seller, maintain the average selling price over the last 10 items sold.*

```
INSERT INTO stdout
SELECT sellerID, AVG(price)
    OVER (PARTITION BY sellerID ROWS 9 PRECEDING)
FROM ClosedPrice;
```

In the next example, we find the first highest bid during the last 10 minutes. To do this, we invoke UDA **maxpair** on a logical window of 10 minutes.

Example 8 *UDA on time-based window: Every 10 minutes return the first highest bid in the recent 10 minutes.*

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

5.1. Defining UDAs on Windows

For most aggregates the efficiency of their computation on windows can be greatly improved by delta maintenance techniques similar to those used to support concrete views. For that, we introduce a new state, called EXPIRE in UDA definition, as illustrated by the following example:

Example 9 *The New Construct* EXPIRE

```
WINDOW AGGREGATE myavg(Next Int) : Real
{ TABLE state(tsum Int, cnt Int);
  INITIALIZE : {
    INSERT INTO state VALUES (Next, 1);
  }
  ITERATE : {
    /* here, the incoming tuple is a new tuple */
    UPDATE state SET tsum=tsum+Next, cnt=cnt+1;
    INSERT INTO RETURN
      SELECT tsum/cnt FROM state;
  }
  EXPIRE: {
    /* here, the incoming tuple is an expired tuple */
    UPDATE state SET tsum=tsum-Next, cnt=cnt-1;
  }
}
```

When a new tuple arrives, the system executes the EXPIRE statements for each tuple in the expired window, before it executes the ITERATE routine for the newly arrived tuple. This ensures that the computation reflects the correct state.

The use of EXPIRE in UDAs represents a significant improvement over the delta maintenance constructs presented in [43], since it produces shorter definitions and a more efficient implementation. In [24], we discuss efficient window implementation when the main memory is limited. Under the assumption that there is only one single window, we prove that Last-In-First-Out (LIFO) algorithm is optimal in terms of the number of page faults. In reality, however, there are multiple continuous queries in the DSMS system, and each could have a window of different size, each processing data at a different rate. We show that an policy called the Longest Forward Distance algorithm is optimal in this situation.

Window maintenance issues are shared by all kinds of aggregates, including built-in aggregates, aggregates defined in procedural languages and UDAs. In our examples, we showed UDAs defined in ESL. However, the same design, in particular the use of an EXPIRE state, apply to UDAs written in procedural languages or languages different from ESL [6]. Corresponding primitives should also be used to handle the windows used by built-in aggregates. Indeed significant performance benefits can be achieved with this approach [24].

Although not discussed here, ESL also supports the slide and tumble constructs [6] for general UDAs.

5.2. Effectiveness of Window UDAs

The following example defines UDA **maxpair**. We have used this UDA in Example 8, where it is applied on the **Bid** stream with a 10-minute time-based window, and returns a stream of maximum bids within each window.

Example 10 *Optimizing maxpair for windows*

```
WINDOW AGGREGATE maxpair(iValue, iPoint) :
                              (mValue, mPoint)
{
    TABLE mpair(value, point);
    INITIALIZE: ITERATE:{
      DELETE FROM mpair WHERE value < iValue;
      INSERT INTO mpair VALUES(iValue, iPoint);
    }
    EXPIRE:{
      DELETE FROM mpair WHERE point = iPoint;
      INSERT INTO RETURN
        SELECT maxpair(value,point) FROM mpair;
    }
}
```

To find the first bid with maximum price in the window, the UDA iterates over the stream and selectively buffers bids that occurred in the last 10 minutes. Note that a bid which is not the highest in the current window could become the highest in the next window, so we must buffer the bids. However, for each incoming bid, we can remove those in the buffer that have been out-bidden by the new bid. Finally we use the UDA **maxpair** called without the window modifier to find the maximum bids from the buffer. In ESL, the window and windowless versions of an aggregate are treated as two different procedures, where the presence of the OVER clause results in the invocation of the procedure for the widow version of the aggregate, and its absence results in the call of the basic windowless version.

How many tuples we need to buffer? It can be shown that, on the average, mpair contains $log_2(N)$ tuples, where N is the number of tuples in the window; the windowless version of **maxpair** takes linear time to the size of **mpair** while the equivalent SQL query without UDA takes quadratic time. Thus, our both versions of **maxpair** are quite efficient.

6. Concrete Views and Joins

All the constructs defined so far are based on SQL:2003. Even UDAs, which were not included in the official standards were contained in early SQL-3 draft, and are now part of several commercial DBMSs that support their definition via external procedural languages (rather than its the native definition supported by ESL).

So far, we have considered queries that operate on data streams and return data streams as their output. We now turn to the situation where we want to define concrete table-like views from data streams. This problem has been studied in [1] where window-like construct similar to those used for aggregates are used for the task. [1]. In ESL we use instead special table function to achieve the same objective: this approach is consistent with SQL:2003 where table functions are can be used to recast practically any kind of external data source into SQL tables [31].

For instance, we can define a window of a 24-hour range on stream ClosedAuction. When there is a new tuple coming to OpenAuction, we join it with tuples in the window:

Example 11 Short Auction Query: *Report all auctions which closed within 24 hours of their opening.*

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

Thus, TABLE (ClosedAuction OVER ...) generates a TABLE-like view of the stream ClosedAuction over a window defined using the usual RANGE|ROWS statements; however PRECEDING now takes an explicit synchronization parameter. In the example, the window is explicitly synchronized with O, specifying that all the tuples up to and including the timestamp of O being joined with the window must be in the window. Thus the implementation of this window requires synchronization policies similar to those used in the union of streams,

discussed earlier. To reduce the overhead involved in these techniques, we allow the ESL programmer to replace PRECEDING O with PRECEDING SELF.

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

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

When PRECEDING SELF is specified, the window spans a period of 24 hours, with computed with respect to its *most recent tuple in the buffer*—with a semantics similar to that used for windows on aggregates. In the current Stream Mill implementation, this means that upon arrival of a new OPENAUCTION tuple, the system process all newly arrived tuples in CLOSEDAUCTION but does not wait until it sees one with timestamp equal or greater than that of O. In a nutshell, 'SELF' falls back to an operational semantics, that only assures synchronization on a best-effort basis and depending on system implementation and workload. For many applications this more relaxed policy is sufficient and preferable since it poses less demands on the system. Data streams with latent timestamps are typically processed using this policy. In fact, if CLOSEDAUCTION had latent timestamps, only the 'SELF' synchronization policy would be allowed in ESL (also only count-based windows would be allowed, whereas both kinds of windows are allowed now since CLOSEDAUCTION is explicitly timestamped.) Likewise, the 'SELF' option is required if OPENAUCTION had a latent timestamp (and the stream produced would also have a latent timestamp in that case.)

Therefore, ESL can support joins of multiples streams either using the union-based approach illustrated in Example 5) and the window-based solution of Example 11, above. In general to compute the join of streams A and B on a given window we instead need to union the results of joining A on a window on B, with the join of B on a window on A. As described in [21] this approach is conducive to efficient implementation and query optimization. No union is needed in the case of self-joins and when event A is known to precede event B as in example 11.

Our next example is to compute a delayed stream:

Example 13 Delayed Stream: *List all calls made 10 minutes ago in:* calls(call_ID, event, time)

```
SELECT call_ID
FROM TickSecond T,
    TABLE(calls OVER 10 MINUTE PRECEDING T) AS W
WHERE T.time - W.time = 10 MINUTE AND event = 'start';
```

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 stream.

Finally observe that, in ESL, there are difference between table-functions windows and aggregate windows inasmuch as in the former (i) an explicit argument after PRECEDING is required, and (ii) the PARTITION BY clause is not supported. The similarities between the two are however such, that Stream Mill can provide a unified and highly optimized support for both.

6.1. Concrete Views

ESL also supports concrete views created as windows on streams using a syntax similar to the windows generated as table expressions in the previous examples (but only the SELF option is allowed here). For instance:

Example 14 Concrete View: *A table containing all the auctions where the price was above 100000 in the last two hours.*

```
CREATE TABLE bigitems AS (SELECT *
        FROM OpendAuction OVER
            (RANGE 2 HOUR PRECEDING SELF)
        WHERE start_price > 100000) IMMEDIATE
```

This view has the semantic property of tables. Therefore, as any other query on database tables, this is an *ad hoc query* that returns results and then quits. Here we support SQL:2003 IMMEDIATE/DEFFERRED options of concrete views. In the first case the system maintains this view up-to-date and responds immediately to the user query. In the second case, the system rebuilds the view when the query is received. In the second case, the response to the query might be slow (in our example, it could take up to 2 hours to rebuild the data) and the results might be returned incrementally, almost as in a data stream. The fact that the users can write query that display a continuum of behavior between data stream and database applications is a positive sign from the practical viewpoint. However, there is no ambiguity in the semantics of the the two kinds of queries in ESL, since

- Continuous queries are those defined by simple SQL statements where data stream appear as an argument in their 'from' clauses (and all the remaining arguments in the from clause are database tables, or equivalent). Union of such statements also define contin-

uous queries. No other statements define continuous queries.
  – Continuous queries persist until they are turned off explicitly by the user.

- Ad hoc queries are defined by statements where all the arguments in their from clause are database tables, or their equivalent
  – Ad hoc queries are turned off once they complete.

No other queries but those are supported in ESL; furthermore blocking operators are not allowed on data streams, although they are fully allowed on tables or their equivalent.

7. Applications of ESL

Sequence Queries Several query languages have been proposed in the past for sequences and time-series [29, 33]. The main motivation of these languages is that most self-join queries are too complex to express and inefficient to implement if they are written in SQL [29, 33]. Take for instance the following query on time-series of daily temperatures: Find a fortnight of raising temperatures. In SQL, this query requires 14 joins; that many joins cannot be supported efficiently, and are also too complex for users to write. Other queries, such as double-bottom queries [33] would require an unbounded number of self-joins, and might not be expressible in SQL. All these queries can however be expressed using UDAs.

In the following example taken from [33], we have a log of web pages clicked by a user, as follows:

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

A user entering the home page of a given site starts a new session that consists of a sequence of pages clicked; for each session number, **SessNo**, the log shows the sequence of pages visited—where a page is described by its timestamp, **ClickTime**, number, **PageNo** and type **PageType** (e.g., a content page, a product description page, or a page used to purchase the item).

The ideal scenario for advertisers is when users (i) see the advertisement page for some item in a content page, (ii) jump to the product-description page with details on the item and its price, and finally (iii) click the 'purchase this item' page. This advertisers' dream pattern can be expressed by the following UDA query, where 'a', 'd', and 'p', respectively, denote an ad page, an item description page, and a purchase page. We store the last three page types using the memo table in the UDA **find_pattern**. Then we issue a self-join query to check the condition. Since the memo table only contains three tuples at all time, the self-join query is very efficient.

Example 15 *Sequence query*

```
INSERT INTO stdout
SELECT SessNo, find_pattern(PageType)
FROM Sessions
GROUP BY SessNO;

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

In [4], we designed a sequence query language called ESL-TS. This query is expressible in ESL-TS with only one statement. The implementation of ESL-TS is based on the constructs discussed in this paper.

In Example 16, we compute the top 5 purchases in each 10-hour window. This is realized by UDA **window_top5**, which iterates over the stream **purchase** and selectively buffers purchases made in the last 10 hours. For each purchase record r, we count the number of future purchases (as they come in) that have a larger amount than r. If there are more than 5 such purchases, we remove r from the window buffer since it will not qualify as one of the top 5 purchases for now or in the future. This memory optimization technique can be gracefully expressed in ESL. Furthermore, the tuple expiration is transparent to users (with EXPIRE absent or empty).

Example 16 *Top 5 purchases within each 10 hour window*[6]

```
CREATE STREAM purchase(id, amount) SOURCE . . .

SELECT id, window_top5(amount,0)
OVER (RANGE 10 HOURS PRECEDING)
FROM purchase;

WINDOW AGGREGATE window_top5(pamount, cnt):{
    INITIALIZE : ITERATE : {
      UPDATE WINDOW W SET W.cnt=W.cnt+1
        WHERE W.amount < amount;
      DELETE FROM WINDOW W WHERE W.cnt ≥ 5;
      INSERT INTO RETURN
        SELECT top5(W.amount) FROM WINDOW W;
    }};
```

6 The windowless version of UDA **top5()** can be found at [11]

Our next example is on-line mining — one of the key applications of data streams. The task is to learn an accurate classifier from an unbounded data stream with concept-drifts [18, 39].

In order to maintain a classifier that represents the most up-to-date concept of the underlying dataset, we must learn from new data and 'forget' old data. However, both revising the current decision tree and building a new decision tree are time-consuming, and have negative impact on prediction accuracy with concept-drifting data streams [39].

Here, we adopt a more efficient approach: we partition the stream into data chunks of fixed size, and learn a classifier from each of them. We then combine those classifiers that have low prediction errors on the most recent training data to predict the most recent testing data. The above process can be implemented with a UDA **classifystream**.

Example 17 *stream classifier with concept-drifts*

```
AGGREGATE classifystream(col_1, ..., col_n, label):
{    TABLE state(cnt Int) AS VALUES (0);
     INITIALIZE: ITERATE : {
        SELECT learn(W.*) FROM WINDOW AS W
           WHERE ((SELECT cnt FROM state) = 1000
                OR  ((SELECT cnt FROM state) % 1000 = 0 AND
                     (SELECT sum(|classify(W.*)-W.label|)
                     FROM WINDOW AS W
                      WHERE W.label=NULL)≥ threshold))
              AND W.label <> NULL;
        UPDATE state SET cnt=cnt+1;
        INSERT INTO RETURN
           SELECT classify(V.*)
           FROM VALUES(col_1,...,col_n) AS V
           WHERE label = NULL;
     }
}
```

The UDA **classifystream** in Example 17 implements a classifier over a stream with concept-drifts. It makes use of a UDA classify and learn, which are either available as a built-in or can be defined using UDAs as described in [43, 42].

Now, all that is left to do is to apply our **classifystream** on the union of the (labeled) training data and the (unlabeled) testing data:

```
INSERT INTO stdout
SELECT classifystream(S.*)
OVER (ROWS 10000 PRECEDING)
FROM stream AS S;
```

At each point, we must determine whether we should (re-)learn a classifier. We (re-)learn a classifier if (i) we reach the boundary of the first window ($cnt = 1000$); or (ii) at the boundary of every 1000 records ($cnt \% 1000 = 0$), the classification error on the last window exceeds a user-specified error threshold. Here, we use zero-one loss ($|$classify($\star$)-label$|$) to estimate classification error .

To reduce overfitting, we can implement a more sophisticated classifier using classifier ensembles [39].

8. Related Work

The objective of overcoming SQL's limitations in dealing with time series and sequences has motivated significant database research long before the emergence of data streams.

The first line of work focuses on using composite events as triggers of active databases rules [14, 13, 27]. These projects observed that the event-condition statements supported in commercial SQL DBMS are insufficient to detect complex event patterns in time-series, and therefore proposed powerful extensions based on a marriage of SQL with regular expressions and temporal aggregates [14, 13, 27].

In terms of expressive power, composite-event languages support most of the functions needed for processing sequences or data streams, but it is far from clear that the active database architecture can be extended to provide the query performance and high bandwidth required on sequences or data streams. Therefore, most research projects on data sequences have instead used query language extensions (and this is also true for data streams).

Because of space limitations, and the presence of recent surveys [2, 15] we will not attempt to give a comprehensive overview of the large body of excellent research work that is being produced on data streams. We will only focus on some points that are more directly related to ESL.

For instance stream declaration with various kinds of timestamps is extensively discussed in CQL, and the concept of heartbeats is addressed. ESL provides an additional type of timestamp, i.e. latent timestamp. Windows play a critical role in CQL as it is a stream-to-relation operator. CQL uses windows as stream modifiers, unlike the role of aggregate modifiers in SQL:2003 and ESL.

Instead of extending SQL, Aurora defines query plans via a "boxes and arrows" graphical interfaces [6]. Aurora supports eight primitive operations, among which four are windowed operators. It also supports user-defined aggregates, which are defined with semi-procedural constructs rather than SQL. The aggregation operators have optional parameters, making their semantics dependent on stream arrival and processing rates. All operators in Aurora are stream-to-stream, although historical data can be stored into relations via *connection points*. Ad-hoc queries can be issued on relations.

TelegraphCQ proposes a SQL-like language with extension of window constructs. As CQL, TelegraphCQ treats windows as stream modifier rather than aggregate modifiers in ESL and SQL:1999. The usage of aggregates must be combined with windows, since all aggregates in TelegraphCQ are blocking. In [12], a low-level C/C++-like for-loop aiming at more general window, such as backward windows and windows in the past.

GSQL is a pure stream query language designed for Gigascope, a stream database for network applications [9]. GSQL is an SQL-like language allowing query composition and query optimization.

In GSQL[9], stream declaration with timestamp is not discussed. GSQL supports sort-merge union of streams, joins of two streams, and aggregation. The concept of window is absent in GSQL. Since GSQL is a pure stream language, the operations involving tables are missing. Gigascope has the concept of external functions (e.g. to implement algorithms by network analysts) written in procedural languages.[7]

9. Conclusions

An SQL-based query language represents the obvious choice for systems that want to support applications spanning both DB tables and data streams. However, continuous queries on streaming data are so different from traditional queries on stored data that an in-depth analysis of the new requirements is the sine-qua-non for an effective design. Some of the requirements, e.g., to avoid blocking computations, and minimize memory usage, have been recognized by previous research projects, but ESL goes further of other projects in addressing the expressive power and generality issues that follow from these new requirements, and design a complete language for data stream applications, under both a theoretical and practical viewpoint. The completeness of ESL from a theoretical viewpoint was proven in [22] where we showed that the language can express all computable queries on database tables, and all the queries expressible using nonblocking query operators (i.e., all monotonic queries) on data streams. In this paper, we have instead elucidated the practical language design issues, and showed that ESL can handle advanced applications, such as approximate computations, sequence queries, and data stream mining, that are well beyond the capabilities of other data stream systems. Furthermore, we have proposed a design that (i) minimizes the deviations from the SQL:2003 standards and (ii) integrates continuous data stream queries, and ad-hoc database queries with a clear semantics. The key extensions with respect to SQL:2003 are (a) a sort-merge semantics for union, (b) nonblocking UDAs (c) window-based extensions of such UDAs, and (c) special table function to create table-like windows on streams.

7 In ESL, we also support external functions written in procedural languages as well as UDAs written in SQL.

While topic (b) was discussed in previous papers, the remaining topics are new.

We have shown that the limited set of syntactic extensions added to SQL:2003 by ESL allows users to express succinctly and efficiently complex queries, including continuous mining queries and all the queries in [3]. Moreover, the use of UDAs can allow significant savings in memory and computations in many situations—particularly when used as a replacement for self-joins. Due to space limitations, we leave for later papers a discussion of the architecture and implementation of the Data Stream Mill system, which features a new data stream manager and continuous-query manager—along with the database manager and query manager supporting SQL and blocking and nonblocking UDAs.

The current version of the ESL system is available from [11].

Acknowledgements

We are grateful to Hetal Thakkar and Yijian Bai for their comments and help in the Data Stream project.

References

[1] 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.

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

[3] Shivnath Babu. Stream query repository. Technical report, CS Department, Stanford University, http://www-db.stanford.edu/stream/sqr/, 2002.

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

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

[6] 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.

[7] S. Chandrasekaran and M. Franklin. Streaming queries over streaming data. In *VLDB*, 2002.

[8] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD*, pages 379–390, May 2000.

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

[10] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows:

(extended abstract). In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 635–644. Society for Industrial and Applied Mathematics, 2002.

[11] http://wis.cs.ucla.edu/projects/stream-mill/.

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

[13] S. Gatziu and K.R. Dittrich. Events in an object-oriented database system. In *Proceedings of the First Intl. Conference on Rules in Database Systems*, 1993.

[14] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model and implementation. In *Proceedings of the 18th VLDB*, 1992.

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

[16] J. Han, Y. Fu, W. Wang, K. Koperski, and O. R. Zaiane. DMQL: A data mining query language for relational databases. In *Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD)*, pages 27–33, Montreal, Canada, June 1996.

[17] J. M. Hellerstein, P. J. Hass, and H. J. Wang. Online aggregation. In *SIGMOD*, 1997.

[18] G. Hulten, L. Spencer, and P. Domingos. Mining time-changing data streams. In *SIGKDD*, pages 97–106, San Francisco, CA, 2001. ACM Press.

[19] T. Imielinski and A. Virmani. MSQL: a query language for database mining. *Data Mining and Knowledge Discovery*, 3:373–408, 1999.

[20] H. Jagadish, I. Mumick, and A. Silberschatz. View maintenance issues for the chronicle data model. In *PODS*, pages 113–124, 1995.

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

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

[23] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE TKDE*, 11(4):583–590, August 1999.

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

[25] Sam Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, pages 49–61, 2002.

[26] R. Meo, G. Psaila, and S. Ceri. A new SQL-like operator for mining association rules. In *VLDB*, pages 122–133, Bombay, India, 1996.

[27] I. Motakis and C. Zaniolo. Temporal aggregation in active databases. In *Int. Conf. on the Managment of Data*, May 1997.

[28] Chang-Shing Perng and D. S. Parker. Sql/lpp: A time series extension of sql based on limited patience patterns. In *Database and Expert Systems Applications, 10th Int. Conference, DEXA '99*, volume 1677 of *Lecture Notes in Computer Science*. Springer, 1999.

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

[30] R. Ramakrishnan, D. Donjerkovic, A. Ranganathan, K. Beyer, and M. Krishnaprasad. Srql: Sorted relational query language, 1998.

[31] 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.

[32] Reza Sadri, Carlo Zaniolo, and Amir M. Zarkesh andJafar Adibi. A sequential pattern query language for supporting instant data minining for e-services. In *VLDB*, pages 653–656, 2001.

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

[34] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. In *SIGMOD*, 1998.

[35] P. Seshadri. Predator: A resource for database research. *SIGMOD Record*, 27(1):16–20, 1998.

[36] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. Sequence query processing. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 430–441. ACM Press, 1994.

[37] M. Sullivan. Tribeca: A stream database manager for network traffic analysis. In *VLDB*, 1996.

[38] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *SIGMOD*, pages 321–330, 6 1992.

[39] Haixun Wang, Wei Fan, Philip S. Yu, and Jiawei Han. Mining concept-drifting data streams using ensemble classifiers. In *SIGKDD*, 2003.

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

[41] Haixun Wang and Carlo Zaniolo. Extending sql for decision support applications. In *Proceedings of the 4th Intl. Workshop on Design and Management of Data Warehouses (DMDW)*, pages 1–2, 2002.

[42] 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.

[43] 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.