ATLaS: a Turing-Complete Extension of SQL for Data Mining Applications and Streams

Haixun Wang
IBM T.J. Watson Research Center
Hawthorne, NY 10532
haixun@us.ibm.com

Carlo Zaniolo Richard C. Luo
Computer Science Department, UCLA
Los Angeles, CA 90095
{zaniolo|lc}@cs.ucla.edu

1. INTRODUCTION

ATLaS is a powerful database language and system that enables users to develop complete data-intensive applications in SQL—by writing new table functions and aggregates in SQL, rather than in procedural languages as in current O-R systems. As a result, ATLaS is Turing-complete [12], and very suitable for advanced data-intensive applications, such as data mining and stream queries [10]. The first ATLaS system and application suite is available for downloading from [1]. The suite includes several data mining functions coded in ATLaS' SQL that run with only a modest (20–40%) performance overhead with respect to the same applications written in C/C++. We are now developing an extension of ATLaS for streams. Our proposed ACM SIGMOD 2003 demo will illustrate the key features and applications of ATLaS. In particular, we will demonstrate how to:

- write new aggregates and table functions in SQL,
- define count-based and time-span based windows on streams,
- perform efficient delta-based maintenance of aggregates on such windows,
- use these constructs to code concisely complex applications, including, data mining functions, approximate queries, and classifiers maintained over data streams.

2. DATABASE APPLICATIONS IN ATLAS

User Defined Aggregates (UDAs) are important for decision support, stream queries and other advanced database applications [9, 2, 4]. ATLaS adopts from SQL-3 [6] the idea of specifying a new UDA by an *initialize*, an *iterate*, and a *terminate* computation; however, ATLaS let users express these three computations by a single procedure written in SQL [8]— rather than three procedures coded in procedural languages as in SQL-3[1].

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

Example 1 defines an aggregate equivalent to the standard **avg** aggregate in SQL. The second line in Example 1 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 labelled 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. This is done by the 'INSERT INTO RETURN' statement that, in the example, returns a single value but, in general, could return several values[2].

EXAMPLE 1. *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 online aggregation [5], an important concept not considered in SQL-3. Since averages converge to a final value well before all the tuples in the

[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

set have been visited, we can have an online aggregate that returns the average-so-far every, say, 200 input tuples. In this way, the user or the calling application can stop the computation as soon as convergence is detected. Then we can write the UDA of Example 2, where the RETURN statements appear in ITERATE instead of TERMINATE. The UDA **online_avg**, so obtained, takes a stream of values as input and returns a stream of values as output (one every 200 tuples). While each execution of the RETURN statement produces here only one tuple, in general, it can produce (a stream of) several tuples. Thus UDAs operate as general stream transformers. Observe that the UDA in Example 1 is blocking, while that of Example 2 is nonblocking. Thus, nonblocking UDAs are easily expressed in ATLaS, and clearly identified by the fact that their TERMINATE clauses are either empty or absent.

The typical default semantics for SQL aggregates is that the data is first sorted according to the GROUP-BY attributes: thus the very first operation in the computation is a blocking operation. Instead, ATLaS uses a (nonblocking) hash-based implementation for the GROUP-BY calls of the UDAs[3]. This default operational semantics lead to a stream oriented execution, whereby the input stream is pipelined through the operations specified in the INITIALIZE and ITERATE clauses: the only blocking operations (if any) are those specified in TERMINATE, and these only take place at the end of the computation.

EXAMPLE 2. *Online averages*

```
AGGREGATE online_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;
        INSERT INTO RETURN
           SELECT sum/cnt FROM state
           WHERE cnt % 200 = 0;
     }
     TERMINATE : {  }
}
```

UDAs are called as any other builtin aggregate. For instance, given a database table employee(Eno, Name, Sex, Dept, Sal), the Example 3 computes the average salary of employees in department 1024 by their gender. The first two lines in Example 3 illustrate that ATLaS can access tables stored in external sources, such as Berkely DB files [7]. The next two lines of Example 3 filter the tuples from the **employee** table using the condition **Dept= 1024**; the tuples that survive the filter are then pipelined to the aggregate **online_avg**.

EXAMPLE 3. *UDAs and Tables in ATLaS*

```
TABLE employee(Eno Int, Name Char(18), Sal Real,
        Dept char(6)) source 'C:\mydb\employees';
SELECT Sex, online_avg(Sal)
FROM employee WHERE Dept=1024 GROUP BY Sex;
```

In Example 4, we define a **minpairs** aggregate that returns the point where a minimum occurs along with its value at the minimum.

[3]However, sort-based UDAs can also be specified in ATLaS, e.g., using ORDER BY clauses [10]

EXAMPLE 4. *Define* **minpairs** *in ATLaS*

```
AGGREGATE minpairs(iValue Int, iTime Timestamp)
          : (mValue Int,mTime Timestamp)
{    TABLE mvalue(value Int);
     TABLE mpoints(pointsInt);
     INITIALIZE: {
        INSERT INTO mvalue VALUES (iValue);
        INSERT INTO mpoints VALUES(iTime);
     }
     ITERATE: {
        UPDATE mvalue SET value = iValue
           WHERE iValue < value;
        DELETE FROM mpoints WHERE SQLCODE = 0;
        INSERT INTO mpoints
           SELECT iTime FROM mvalue
           WHERE iValue =mvalue.value;
     }
     TERMINATE: {
        INSERT INTO RETURN
           SELECT value, pointFROM mpoints, mvalue;
     }
}
```

We use two tables: the **mvalue** table holds, as its only entry, the current min value, while **points** holds all the points in time where this value occurs. In the ITERATE statement we have used the SQLCODE to 'remember' if the previous statement updated **mvalue**, since if the old value was larger than the new one then the old points must be discarded. Indeed, SQLCODE is a convenient labor-saving device of standard SQL that is set to 0 if the last SELECT, UPDATE, or DELETE statement was *successful*—i.e., if more than zero tuples were, respectively, selected, updated, or deleted by its execution. Then, the last statement in ITERATE adds the new **iTime** to **mpoints** if the input value is equal to the current min value. Observe that the formal parameters of the UDA function are treated as constants in the SQL statements. Thus, the INSERT statement in ITERATE adds the constant **iTime** into the **mpoints** relation, provided that **iValue** is the same as the value in **mvalue**. Therefore, the FROM and WHERE clauses operate here as conditionals. The RETURN statement in TERMINATE returns the final list of min pairs as a stream.

ATLaS also supports the definition of table functions in SQL, which have proven very useful in many applications [9, 10]. Because of space limitations, however, we will cut their discussion short, and instead concentrate on the novel ways in which ATLaS UDAs deal with streams and windows.

3. STREAM APPLICATIONS IN ATLAS

There is much current research on stream queries, which use approximate aggregates, synopses, and sliding windows [2, 11, 3]. ATLaS supports windows in the FROM clause along the lines proposed in [2]. Thus, a window specification consists of:

1. an optional partitioning clause, which partitions the data into several groups and maintains a separate window for each group,
2. a window size, using either the count of the elements in the window, or the range of time covered by the window (i.e., its time-span).
3. an optional filtering predicate.

In a stream of of telephone calls records [2], we want to compute the average call length, but considering only the ten most recent long-distance calls placed by each customer. Then we can write the following query [2]:

EXAMPLE 5. *Count-Based Window on a Stream*

```
STREAM calls(customer_id Int, type Char(6), minutes Int,
             Tstamp: Timestamp)  SOURCE mystream;
SELECT AVG(S.minutes)
FROM Calls S [  PARTITION BY S.customer id
                ROWS 1000 PRECEDING
                WHERE S.type = 'Long Distance']
```

where the expression in braces defines a sliding window on the stream of calls. The meaning of this query is that for each new long-distance tuple coming in the stream, the average of this and the previous 999 tuples is computed and returned to the user. Thus this query receives a stream as input and generates a stream as output. The input stream is specified in ATLaS using the SOURCE constructor; then the system adds to each tuple in the input stream a (transaction time) timestamp, under the attribute **Tstamp**. These timestamps are used to support time-span based windows, which can be specified as in the following example [2]:

EXAMPLE 6. *Timestamp-based window*

```
STREAM calls(customer_id Int, type Char(6), minutes Int,
             Tstamp: Timestamp)  SOURCE mystream;
SELECT AVG(S.minutes)
FROM Calls S [  PARTITION BY S.customer id
                RANGE 5 MINUTES PRECEDING
                WHERE S.type = 'Long Distance']
```

Here the window for computing averages is 5 minutes. Thus when a new tuple arrives with timestamp t, all tuples with timestamp less than $t - 5$ *minutes* are dropped from the buffer (a queue), and then the average is recomputed on the new set of tuples so derived, and the result is appended to the output stream. Recomputing the aggregate on all values in the window is useful to specify the semantics of windows, but is too inefficient for a concrete semantics. Indeed, for a window of size 1000, we do not have to recompute the average of all 1000 tuples for each new tuple streaming in: it suffices to add to the old total the value in the new tuple, subtract the values of the expired tuples, and return the result divided by the current count (which remains constant in a count-based window, but can change in a time-span based window). This delta-based computation of aggregates on windows is fully supported by ATLaS, as shown in Example 7. UDAs on windows are defined using the three states INITIALIZE, ITERATE, and REVISE (which replaces TERMINATE). The first two states are active in the transient situation, when the query is first started on the stream, and the boundary of the window have not yet been reached. Once the boundary of the window have been reached, then ITERATE is no longer true, and every new incoming tuple is processed by REVISE. In this state, the system maintains the table EXPIRED holding the input tuples that just expired (one for count-based windows, zero, one, or many for time-span based windows). This table has the same schema as the input tuples, (i.e., EXPIRED(**Next Int**) for Example 7), and it is updated automatically by the system. Thus,

EXAMPLE 7. *Defining* **myavg** *on windows*

```
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;
     }
     REVISE : {
        UPDATE state SET
           tsum=tsum + Next -
                  sum(SELECT Next FROM EXPIRED),
           cnt=cnt+1-count(SELECT * FROM EXPIRED);
        INSERT INTO RETURN
           SELECT tsum/cnt FROM state;
     }
}
```

the sum and the count of the tuples in EXPIRED can now be used to update the sum and the count, and then return the average value of the window. The presence of REVISE defines **myavg** in Example 7 as a window aggregate. Thus, when **myavg** is applied to tuples of a FROM clause with window, such as in Examples 5 and 6, ATLaS uses the UDA of Example 7; otherwise it uses that of Example 1.

The suggested delta-based extension of UDAs for windows is general and powerful. For instance, to extend the **minpairs** aggregate of Example 4 to handle windows we can write:

EXAMPLE 8. **minpairs** *for windows*

```
AGGREGATE minpairs(iValue Real, iTime Timestamp):
          (mValue Int, mTime Timestamp)
{    TABLE dpairs(dValue Int, dTime Timestamp);
     INITIALIZE: {
        INSERT INTO dpairs VALUES (iValue, iTime);
     }
     ITERATE: {
        DELETE FROM dpairs
           WHERE iValue < dValue;
        INSERT INTO dpairs VALUES (iValue, iTime);
     }
     REVISE: {
        DELETE FROM dpairs WHERE
           iValue < dValue OR
           (dValue, dTime) IN (SELECT * FROM EXPIRED);
        INSERT INTO dpairs VALUES(iValue, iTime);
        INSERT INTO RETURN
           SELECT minpairs(dValue, dTime)
           FROM dpairs;
     }
}
```

Thus, we use **dpairs** to hold the dominant pairs—i.e., those for which there is no younger tuple in the stream with a smaller value. Then, it can be shown that, on the average, **dpairs** contains $log_2(N)$ tuples, where N is the number of tuples in the window; thus, our window version of **minpairs** is quite efficient. Moreover, with no window specified in its FROM clause, Example 8 calls **minpairs** of Example 4 to perform the actual computation on table **dpairs**. The two examples are actually compiled as different UDAs; thus, no real recursion is involved here.

Besides windows, a wide range of aggregate modifiers is useful in dealing with the continuous aggregates used in stream

computations. On line UDAs, often fall into this group. For instance, we can easily modify Example 2 to make it work on streams. For example, if we want the average of the last 200 values returned every 200 tuples, we only need to reset to zero **cnt** and **tsum** in the ITERATE statement. If instead we want an effect similar to a window, but in more gradual manner, is sufficient to combine the last average with the new incoming value, assigning to this a slightly larger weight. In a nutshell, ATLaS makes available to users the wide variety of approximate, and application-specific aggregates needed for stream applications. Due to space limitations, here we have only discussed simple application examples, but complex applications, including decision tree classifiers, association rules, and other data mining functions can be concisely written and efficiently implemented in SQL using ATLaS [8, 9, 10].

4. THE ATLAS SYSTEM

The ATLaS system consists of two main components: (a) the database storage manager, and (b) the language processor. The database storage manager consists of:

1. The Berkeley DB library [7] that ATLaS uses to support access methods such as the B^+Tree, and Extended Linear Hashing on disk-resident data, and
2. access methods including in-memory tables, R^+-trees, and sequential text files, which we added keeping the same APIs as Berkeley DB. While R^+-trees are handy with spatio-temporal queries, in-memory tables are critical for implementing efficiently in ATLaS tries, priority queues, and other in-memory structures used in a variety of algorithms—including data-mining, and graph-optimization algorithms [10]

The main data structure used in the language processor is the query graph; the nodes of the query graph represent operations, such as `SELECT`, `INSERT` and `DELETE`, and the arcs represent parent nodes consuming the data streams produced by their children nodes. After the parser builds the initial query graph, this is transformed by a rewriter module which performs various optimization steps, such as predicate push-up/push-down, UDA optimization, index selection, and in-memory table optimization. While ATLaS performs sophisticated local query optimization, it does not attempt to change the global execution plan, which therefore remains under programmer's control. After the rewriting, the code generator translates the query graphs into C++ code, which is then compiled and linked with the database storage manager and external library and user-defined functions.

The runtime model of ATLaS is based on data pipelining. All UDAs, including recursive UDAs, are pipelined, and tuples inserted into the RETURN relation during the INITIALIZE/ITERATE steps are returned to their caller immediately. Therefore, local variables (temporary tables) declared in a UDA can not reside on the stack. Instead, they are kept in an internal structure which retains the values of local variables between successive INITIALIZE/ITERATE/TERMINATE calls.

ATLaS hash-based implementation of UDAs works well for the PARTITION construct in windows. However, in implementing the window construct for streams, the cost of storing the window tuples, and computing those just expired raised serious concerns. Our approach is to manage records in active windows using disk files, where the extents of the logical windows are maintained as disk offsets. A PARTITION clause denotes multiple logical windows, one for each partition key, an thus, it can be expensive to support when the key has many values. Therefore, we cluster records for partitioned windows by their keys, and store them in the same disk file.

ATLaS supports data sharing among multiple queries that access the same external data stream concurrently. A single procedure is responsible for reading the data from the external stream and delivering them to the disk buffers of each individual query. Furthermore, window specifications of different queries can share disk buffers if they have the same filtering predicates and PARTITION clause.

The most recent version of ATLaS, and reports on current developments, can be downloaded from [1].

5. REFERENCES

[1] ATLaS home: `http://wis.cs.ucla.edu/atlas`

[2] Brian Babcock, Shivnath Babu, Rajeev Motwani, Jennifer Widom. Models and Issues in Data Streams, PODS 2002, 1-16.

[3] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. ACM SIGMOD 2000, 379-390.

[4] P. Domingos and G. Hulten: Mining high-speed datastreams. In Proc. of the 2000 ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining, 71-80.

[5] J. M. Hellerstein, P. J. Haas, H. J. Wang: Online Aggregation. *SIGMOD, 1997.*

[6] ISO/IEC JTC1/SC21 N10489, ISO//IEC 9075: Committee Draft (CD), Database Language SQL, July 1996.

[7] Sleepycat Software, "The Berkeley Database (Berkeley DB)", `http://www.sleepycat.com`.

[8] Haixun Wang and Carlo Zaniolo: Using SQL to Build New Aggregates and Extenders for Object-Relational Systems. VLDB 2000.

[9] Haixun Wang, Carlo Zaniolo: Extending SQL for Decision Support Applications. DMDW 2002.

[10] Haixun Wang, Carlo Zaniolo: ATLaS: A Native Extension of SQL for Data Mining and Stream Computations UCLA CS Dept, Technical Report, `http://wis.cs.ucla.edu/publications.html`.

[11] Sam Madden, Mehul A. Shah, Joseph M. Hellerstein, Vijayshankar Raman: Continuously Adaptive Continuous Queries over Streams. SIGMOD 2002, 49-61.

[12] Yan-Nei Law, Haixun Wang, Carlo Zaniolo: Blocking, Monotonicity, and Turing Completeness in a Database Language for Sequences and Streams, UCLA CS Dept. Technical Report, `http://wis.cs.ucla.edu/publications.html`.