ATLAS: a Small but Complete SQL Extension for Data Mining and Data Streams

[1]Haixun Wang [2]Carlo Zaniolo [2]Chang Richard Luo

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

[2]Computer Science Dept, UCLA
{zaniolo,lc}@cs.ucla.edu

1 Introduction

DBMSs have long suffered from SQL's lack of power and extensibility. We have implemented ATLAS [1], a powerful database language and system that enables users to develop complete data-intensive applications in SQL—by writing new aggregates and table functions in SQL, rather than in procedural languages as in current Object-Relational systems. As a result, ATLAS' SQL is Turing-complete [7], and is very suitable for advanced data-intensive applications, such as data mining and stream queries. The ATLAS system is now available for download along with a suite of applications [1] including various data mining functions, that have been coded in ATLAS' SQL, and execute with a modest (20–40%) performance overhead with respect to the same applications written in C/C++. Our proposed demo will illustrate the key features and applications of ATLAS. In particular, we will demonstrate:

- ATLAS' SQL features, including its native support for user-defined aggregates and table functions.
- Advanced database applications supported by ATLAS' SQL, including continuous queries on data streams and data mining applications such as classifiers maintained over concept-drifting data streams.
- The ATLAS system, including its architecture, query rewriting and optimization techniques, and the data stream management module.

2 ATLAS' SQL

ATLAS adopts from SQL-3 the idea of specifying user defined aggregates (UDAs) by an *initialize*, an *iterate*, and a *terminate* computation; however, ATLAS let users express these three computations by a single procedure written in SQL [4]— rather than three procedures coded in procedural languages as in SQL-3.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003

The standard **avg** aggregate in SQL can be easily defined in ATLAS (Example 1). It uses a local table, **state**, to keep the sum and count of the values processed so far. While, for 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 (see later examples). 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 table by adding the new input value to the sum and 1 to the count. The TERMINATE statement returns the final result(s) of computation by INSERT INTO RETURN (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):

Example 1 *Defining the standard 'avg' aggregate*

```
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;
     }
}
```

This approach to aggregate definition is very general. For instance, say that we want to support online aggregation [3], an important concept not considered in SQL-3. Since averages converge to a final value well before all the tuples in the 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. This gives 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 whether their TERMINATE clause is either empty or absent.

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 : { }
}
```

Straightforward as it appears to be, ATLAS' approach to aggregate definition greatly improves the expressive power and extensibility of SQL. For instance, in ATLAS, UDAs can call other UDAs, including themselves. This enables us to compute, for example, the transitive closure of a graph using a UDA that performs a depth-first traversal of the graph by recursively calling itself. In fact, we proved that SQL so extended is Turing-complete on database tables [7]. As a result, we can express in ATLAS advanced applications such as data mining that are difficult to support well using the current SQL-compliant DBMSs [6].

3 Stream Applications in ATLAS

SQL extensions have been proposed to support continuous queries in SQL [2]. ATLAS' SQL is Turing complete, as a result, stream queries can be implemented in ATLAS without additional language constructs.

ATLAS supports a delta-based computation of aggregates on windows (Example3). UDAs on windows are defined using 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 EXPIRED table 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 3), and it is updated automatically by the system. Thus, 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.

Example 3 *Defining avg 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(E.Next),
               cnt=cnt+1-count(E.*)
            FROM EXPIRED AS E;
         INSERT INTO RETURN
            SELECT tsum/cnt FROM state
      }
}
```

ATLAS also supports a window specification 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.

Thus, to compute the average call length, but considering only the ten most recent long-distance calls placed by each customer we will write the following query:

Example 4 *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 10 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 9 tuples is computed and returned to the user. Thus this query receives a stream as input and generate a stream as output.

4 Data Mining Applications in ATLAS

Using table functions and recursive aggregates, Algorithm 1 implements a scalable decision tree classifier using merely 14 SQL statements.

Algorithm 1 A Scalable Decision Tree Classifier

```
 1: AGGREGATE classify(iNode Int, RecId Int, iCol Int,
                       iValue Int, iYorN Int)
 2: { TABLE treenodes(RecId Int, Node Int, Col Int,
                      Value Int, YorN Int);
 3:   TABLE mincol(Col Int);
 4:   TABLE summary(Col Int, Value Int, Yc Int, Nc Int)
                      INDEX (Col,Value);
 5:   TABLE ginitable(Col Int, Gini Int);
 6:   INITIALIZE : ITERATE : {
 7:     INSERT INTO treenodes
          VALUES(RecId, iNode, iCol, iValue, iYorN);
 8:     UPDATE summary
          SET Yc=Yc+iYorN, Nc=Nc+1-iYorN
          WHERE Col = iCol AND Value = iValue;
 9:     INSERT INTO summary
          SELECT iCol, iValue, iYorN, 1-iYorN
          WHERE SQLCODE<>0;
      }
10:   TERMINATE : {
11:     INSERT INTO ginitable
          SELECT Col, sum((Yc*Nc)/(Yc+Nc))/sum(Yc+Nc)
          FROM summary GROUP BY Col;
          HAVING count(Value) > 1
             AND sum(Yc)>0 AND sum(Nc)>0;
12:     INSERT INTO mincol
          SELECT minpair(Col, Gini)→mPoint
          FROM ginitable;
13:     INSERT INTO result
          SELECT iNode, Col FROM mincol;
        {Call classify() recusively to partition each of its}
        {subnodes unless it is pure.}
14:     SELECT classify(t.Node*MAXVALUE+m.Value+1,
                  t.RecId, t.Col, t.Value, t.YorN)
          FROM  treenodes AS t,
            ( SELECT tt.RecId RecId, tt.Value Value
              FROM treenodes AS tt, mincol AS m
              WHERE tt.Col=m.Col) AS m
          WHERE t.RecId = m.RecId
          GROUP BY m.Value;
      }
    }
```

A detailed description of Algorithm 1 can be found in [6]. In summary, the INITIALIZE and ITERATE routine of UDA **classify** updates the class histogram kept in the **summary** table for each column/value pair. The TERMINATE routine first computes the gini index for each column using the histogram. If a column has only one distinct value (**count(Value)**$\leq$ **1**), or tuples in the partition belongs to one class (**sum(Yc)=0** or **sum(Nc)=0**), then the column is not splittable and hence, not inserted into **ginitable**. On line 12, we select the splitting column which has the minimal gini index. A new sub-branch is generated for each value in the column. The UDA **minpair** returns the minimal gini index as well as the column where the minimum value occurred. After recording the current split into the **result** table, we call the classifier recursively to further classify the subnodes. On line 14, GROUP BY **m.Value** partitions the records in **treenodes** into **MAXVALUE** subnodes, where **MAXVALUE** is the largest number of different values in any of the table columns. The recursion terminates if table **mincol** is empty, that is, there is no valid column to further split the partition.

UDA **classify** can be applied to relational training sets after they are transformed on the fly to a stream of column/value pairs. Such transformations can be carried out by ATLAS' table functions, which also play a critical role in extending the expressive power of SQL [6].

Due to space limitations, here we have only discussed the classification algorithm, but more complex applications, including the Apriori algorithm for association rule mining, DBSCAN for density based clustering, and other data mining functions can be concisely written and efficiently implemented in SQL using ATLAS [4, 5, 6].

5 The ATLAS System

The ATLAS system consists of the following components: (i) the database storage manager, (ii) the language processor, and (iii) the data stream management engine.

The database storage manager consists of (i) the Berkeley DB library and of (ii) additional access methods including in-memory database tables with hash-based indexing, R^+-tree for secondary storage, sequential text files, etc. We use Berkeley DB to support access methods such as the B^+Tree, and Extended Linear Hashing on disk-resident data. R^+-trees are introduced to support spatio-temporal queries, and in-memory tables are introduced to support the efficient implementation of special data structures, such as trees or priority queues, that are needed to support efficiently specialized algorithms, such as Apriori or greedy graph-optimization algorithms.

The ATLAS language processor translates ATLAS programs into C++ code, which is then compiled and linked with the database storage manager and user-defined external functions. The core data structure used in the language processor is the query graph. The parser builds initial query graphs based on ATLAS' abstract syntax tree. The rewriter, which makes changes to the query graphs, is a very important module, since much optimization, such as predicate push-up/push-down, UDA optimization, index selection,

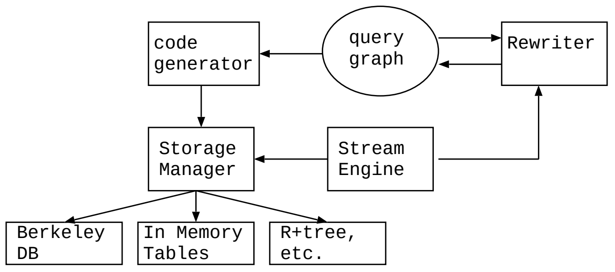

Figure 1: The ATLAS Architecture

and in-memory table optimization, is carried out during this step. While ATLAS performs sophisticated local query optimization, it does not attempt to perform major changes in the overall execution plan, which therefore remains under programmer's control. After rewriting, the code generator translates the query graphs into C++ code.

The runtime model of ATLAS is based on data pipelining. In particular, all UDAs, including recursive UDAs that call themselves, are pipelined; thus, tuples inserted into the RETURN relation during the INITIALIZE/ITERATE steps are returned to their caller immediately. Therefore, local tables declared in a UDA can not reside on the stack. Instead, they are assembled into a `state` structure which is then passed to the UDA for each INITIALIZE/ITERATE/TERMINATE call, so that these internal data are retained between calls.

The data stream management engine is responsible for efficiently maintaining records in windows and the EXPIRED table. Records in a window are stored in a disk file. Count-based windows have fixed sizes, while time-based windows may require dynamic allocation of disk buffers. A window specification with a PARTITION clause may correspond to multiple windows, one for each unique partition key. Records of the windows are clustered by the partition key and stored in a same disk file. ATLAS also 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 the specifications have the same filtering predicate and PARTITION clause.

6 About the Demo

Our demonstration consists of the following parts.

ATLAS Language Features

The only extension introduced by ATLAS is the ability of defining UDAs and table functions in SQL, yet this minimalist approach makes SQL Turing complete. We demonstrate ATLAS' expressive power and ease-of-use through a suite of UDAs and table functions that implement, for example, temporal databases operators (e.g. coalescing), greedy graph-optimization algorithms (e.g. the shortest path algorithm), OLAP operators (e.g. ROLLUP, CUBE), and many others.

ATLAS provides a GUI IDE for writing ATLAS programs. An ATLAS program may consist of several modules, which are either source code in ATLAS' SQL, source code in C/C++, or libraries to be dynamically linked with other modules.

Continuous Queries on Data Streams

We showcase ATLAS' ability of handling continuous queries using the schema and queries in Stanford's Stream Query Repository (http://www-db.stanford.edu/stream/sqr). We implement each query in two different approaches. The first approach uses ATLAS' windows-on-streams construct similar to that proposed in [2]. But ATLAS is Turing complete without this construct. The second approach implements the same query using ATLAS' UDAs and table functions. We will compare the pros and cons of the two approaches.

Data Mining Applications

Substantial extensions have been added to SQL over the years, yet data mining applications remain an unsolved challenge for DBMSs. We demonstrate how ATLAS' minimalist approach attacks this problem. The demonstration package consists of data mining applications such as the decision tree classifier, the Apriori algorithm for association rule mining, and the density-based clustering algorithm DBSCAN. All applications are written entirely in ATLAS' SQL, and can be applied directly on relational data [6].

Recently, stream data mining has been a field of intense research. The demonstration also includes a stream classifier that handles time-changing drifts in the streaming data. It is realized by leveraging ATLAS' continuous-query and data mining ability.

ATLAS incurs only a modest performance overhead with respect to the same applications written in C/C++.

The ATLAS System

The demonstration will reveal ATLAS' internal architecture by focusing on several of its key components, such as the query rewrite module, the query plan generation module, and the stream data management engine.

References

[1] `http://wis.cs.ucla.edu/atlas` ATLAS Homepage

[2] Brian Babcock, Shivnath Babu, Rajeev Motwani, and Jennifer Widom. "Models and Issues in Data Streaming Systems", *PODS 2002*.

[3] J. M. Hellerstein, P. J. Haas, and H. J. Wang. "Online Aggregation". *SIGMOD, 1997*.

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

[5] Haixun Wang and Carlo Zaniolo. "Extending SQL for Decision Support Applications". DMDW 2002.

[6] Haixun Wang and Carlo Zaniolo. "ATLAS: A Native Extension of SQL for Data Mining and Stream Computations". SIAM Data Mining, May 2003.

[7] Yan-Nei Law, Haixun Wang, and Carlo Zaniolo. "On the Properties of a Native Extension of SQL for Data Streams and Data Mining". Submitted for review.