Database System Extensions for Decision Support: the AXL Approach

Haixun Wang
Computer Science Department
University of California at Los Angeles
hxwang@cs.ucla.edu

Carlo Zaniolo
Computer Science Department
University of California at Los Angeles
zaniolo@cs.ucla.edu

Abstract

Research on database-centric data mining is seeking to improve the effectiveness of database systems in decision support applications. Different solutions are now used for different problems, including (i) SQL extensions for more complex OLAP queries, (ii) new datablades for special data types such as time-series, and (iii) architectural extensions to support data mining functions. Here, we proposed a unified solution for all these problems; the solution is based on User-Defined Aggregates (UDAs) expressed in an SQL-like language called AXL. In this paper, we discuss the architecture and implementation of the AXL prototype and its use and performance in expressing data mining functions and complex OLAP queries.

1 Introduction

The difficult problem of extending database systems for decision support applications is of practical importance and of great research interest. Patchwork solutions were proposed in the past for different problems. For instance, various datablades (a.k.a. DB extenders, cartridges and snap-ins) were added to O-R databases for applications such as time-series analysis and text mining. On the other hand, the need for supporting OLAPs in databases was answered by extending SQL with more powerful aggregates defined by new group-by constructs such as data cubes and supergroups [3, 1]. New extensions, such as the window construct, are now being included in the SQL standards [18].

The issue on how to tightly integrate data mining functions into DBMSs still remains an open research challenge, with none of the solutions proposed so far gaining widespread support in the commercial world, or even in the research arena. An incomplete list of research approaches so far proposed include the mine rule operator for association rules [25], the operator Mine for generating and queries sets of propositional rules [24], and the classification middleware proposed in [3]. The study presented in [11] eloquently illustrates the difficult technical challenges posed by the problem.

In this paper, we present a novel approach that is effective in all the database-centric decision-support applications previously mentioned. The approach is based on UDAs defined in an SQL-like language called AXL.

2 UDAs in Object Relational Systems

At UCLA, we have developed the SQL-AG [15] system that supports and extends the UDA specifications originally proposed for SQL3 [10]. For instance, say that we have pairs of numbers **(Point, Value)** and we want to determine a minimum point, rather than a minimum value. A nested sub-query is required to express this computation in standard SQL. But, following the approach proposed for SQL3 [9], we can define the **minpoint** aggregate as shown in Example 1.

Therefore, a stream-oriented computation model is assumed; thus, the user must define the three external procedures, under the labels of INITIALIZE, ITERATE and TERMINATE, which, respectively, specify the computation to be performed for the first pair in the stream, for each successive pair, and when the end of stream is detected and the final values of the aggregate must be returned.

EXAMPLE 1. *Defining an UDA to find the point of least value*

```
AGGREGATE minpoint(INT, INT)
RETURNS INT
STATE state_type
INITIALIZE minpoint_single
ITERATE minpoint_multi
TERMINATE minpoint_terminate
```

```
typedef struct state_s {
  long point; long value;
} state_type;
void minpoint_single(long *point, long *value, state_type *s)
{ s→point = point;
  s→value = value;
}
void minpoint_multi(long *point, long *value, state_type *s)
{ if (s→value > *value) {
      s→point = *point;
      s→value = *value; }
}
void minpoint_terminate(state *s, long *result)
{ *result = s→point; }
```

Thus, in our example, the C procedure **minpoint_single** initializes the state to the first **(Point, Value)** pair, **minpoint_multi** resets the state to the new input pair, if its value is less than that of the current state, and **minpoint_terminate** returns the final point (rather than the final value) from **state**.

While traditional SQL2 aggregates are (dependent on repetitions, but) independent of the order in which the data is visited, order can be significant in the stream-oriented specification for aggregates originally proposed by SQL3 [9], and in the OLAP extensions recently added to SQL [18]. For instance, in the presence of ties, our **minpoint** aggregate could return the first such point; i.e., a result that depends on the particular order in which the pairs **(Point, Value)** are visited. Normally, the validity of the algorithm using the aggregate is unaffected by a nondeterministic aggregate result (don't care nondeterminism). In some applications, such as cumulative or moving-windows temporal aggregates used in time-series [22], the fact that the data is sorted by their time stamp is significant. For instance, if instead of global minima, we want local Y-minima in a sequence of (X, Y) pairs, then the pairs should be sorted by X. On the other hand, an online computation of averages relies on the absence of order, since online aggregates work best on non-skewed data sets [8].

A key improvement made by the SQL-AG system developed at UCLA [15] with respect to the original proposal of SQL3 is the support for *early returns*. Early returns are basically results returned during the computation of the aggregate, as needed to support online aggregation [8] and other advanced forms of aggregation discussed later in [17].

Early returns can be supported by allowing the user to add to the aggregate specification a PRODUCE procedure to generate *early returns*, whereas the TERMINATE procedure generates *final returns*. For instance, **myavg_multi** in the example below can be used to evaluate the rate of convergence for the computation of average and return values at regular interval.

EXAMPLE 2. *A UDA with Early Returns*

```
AGGREGATE myavg(value INTEGER)
RETURNS REAL
STATE state_type
INITIALIZE myavg_single
ITERATE myavg_multi
PRODUCE myavg_produce
TERMINATE myavg_terminate
```

This version of **myavg** contains both early returns and final returns. An interesting special case occurs when the aggregate only contains early returns, i.e., the TERMINATE function **myavg_produce** is either missing or it has been replaced by NOP. Then, it can be shown that the aggregate is monotonic with respect to set containment and can therefore be used without restrictions in recursive SQL queries to code optimized graph traversal and BoM applications that had been a problem for databases since their introduction [21, 17].

We evaluated the performance of our SQL-AG UDAs by re-implementing the standard SQL aggregates such as sum and avg as UDAs and comparing their performance against the DB2 builtins. We found that performance penalty is modest in the worst case (20 % overhead in the worst case), and there are many situations where UDAs yield a overall improved performance with respect to the standard builtins.

Decision support systems are designed to infer interesting statistical patterns from data; therefore they provide a fertile application ground for UDAs. The potential uses of UDAs are however limited by the same usability problem experienced by UDFs, which are known to be very hard to program and debug [11]. Obviously, the same problem would also hinder UDAs coded as multiple UDFs, as originally proposed in SQL3.

This inspired our idea of (i) developing a high level language to facilitate the definition of new UDAs, and (ii) making it powerful enough so that whole data mining algorithms can be expressed in the language, and (iii) base the new language on SQL to maximize its usability and achieve traditional database benefits, such as scalability and parallelizability. A first such language called SADL [17] was developed and tested, and various improvements were found to be needed in terms of expressive power and performance. This lead to the design and implementation of AXL discussed next.

3 AXL

We now introduces AXL by examples; a more complete discussion can be found in [16]. Example 3 defines an online version of `myavg` which returns results for every 100 new values. The first line of this aggregate function declares a local table, `state`, to keep (in memory) 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. 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 set the count to 1, and ITERATE updates the table by adding the new input value to the sum and 1 to the count. The TERMINATE statements return the final result of computation by appending the ratio of sum over count to RETURN; for conformity with SQL syntax, RETURN is viewed as a table, and thus an INSERT INTO construct is used. We also add intermediate results from the computation to RETURN tables as part of the ITERATE statements; this eliminates the need for the special PRODUCE construct used in SADL.

EXAMPLE 3. *Return current average for every 100 records*

```
AGGREGATE myavg(Next INT) : REAL
{   TABLE state(sum INT, cnt INT);
    INITIALIZE : {
        INSERT INTO state VALUES (Next, 1);
    }
    ITERATE : {
        UPDATE state SET
                    sum=sum+Next; cnt=cnt+1;
        INSERT INTO RETURN
        SELECT sum/cnt
        FROM state WHERE cnt % 100 = 0;
    }
    TERMINATE : {
        INSERT INTO RETURN
                SELECT sum/cnt FROM state ; }
}
```

The `minpoint` aggregate defined in Example 1 which uses external routines can be expressed in AXL as shown in Example 4.

EXAMPLE 4. *Define* `minpoint` *in AXL*

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

3.0.0.1 Multiple Scans

In order to express the query: *"Show the average salary of senior managers who make 3 times more than the average employees"*, standard SQL requires the use of a sub-query as shown in Example 5 and hence need to scan the `employee` relation twice to produce the answer.

EXAMPLE 5. *Multiple Scans*

```
SELECT  avg(salary)
FROM    employee
WHERE   title = 'senior manager'
   AND  salary > (SELECT 3 * avg(salary)
                  FROM employee)
```

In Example 5, the two aggregations are performed on the same relation. When senior managers are a very small percentage of the employee population, the tuples qualified for the second aggregation are a small fraction of the whole relation, and further more, they have already been visited during the first aggregation. So the second pass on the relation is wasteful, since the senior manager tuples could have been saved during the first pass. Thus, besides computing the average salary of all the employees, UDA `mscan` defined in Example 6 also filters the employees by their title, and records the salaries of senior managers in an auxiliary table `finalstate` (in memory). In the TERMINATE step, the average salary of senior managers is computed on the `finalstate` table, thus avoiding going through the entire `employee` relation again. In this example, the same SQL statements are performed in the INITIALIZE and ITERATE steps. Also the table allstate is initialized upon its declaration using an SQL query (in this example the query is simply VALUES(0,0)).

EXAMPLE 6. *Using UDA to avoid Multiple Scans*

```
AGGREGATE mscan(title CHAR(20), salary INT) : INT
{
   TABLE allstate(sum INT, cnt INT) AS VALUES(0,0);
   TABLE finalstate(fsalary INT);

   INITIALIZE: ITERATE: {
      UPDATE allstate SET sum=sum+salary,
                          cnt=cnt+1;
      INSERT INTO finalstate VALUES(salary)
         WHERE title = 'senior manager';
   }
   TERMINATE: {
      SELECT avg(fsalary)
      FROM finalstate
      WHERE fsalary > (SELECT 3*sum/cnt
                       FROM allstate);
   }
}
```

Then, Example 5 is simply a SELECT statement using the previous aggregate:

```
SELECT mscan(title, salary) FROM employee;
```

Thus the efficiency of complex computations can often be improved using AXL. The ability of introducing and manipulating new tables is the first cornerstones of AXL's power. The second is the ability of one aggregate calling another, or calling itself recursively.

3.0.0.2 Recursive Aggregates

In AXL, aggregates can call other aggregates. Particularly, an aggregate can call itself recursively. Say we have a relation `children(Parent, Child)`, Example 7 defines a recursive aggregate `alldesc` to find all the descendants of a given person.

EXAMPLE 7. *Offspring*

```
AGGREGATE alldesc(P CHAR(10)) : CHAR(10)
{
   INITIALIZE: ITERATE: {
      INSERT INTO RETURN VALUES(P);
      INSERT INTO RETURN
         SELECT alldesc(Child) FROM children
         WHERE Parent=P;
   }
}
```

Now, we can use the following query to find all the descendants of Tom.

```
SELECT alldesc(Child) FROM children
WHERE Parent='Tom';
```

3.0.0.3 Redefinition

Instead of using INITIALIZE, ITERATE and TERMINATE to define new aggregates, UDAs in AXL can also be defined by building upon aggregates already defined. For instance, the following redefinition groups the sales by territory, then by product, and finally forms a big group consisting of the whole `data` table:

EXAMPLE 8. *Derive new UDA from aggregates already defined*

```
AGGREGATE gssum(territory CHAR(10),
                 product CHAR(10), sales INT):
   (territory CHAR(10), product CHAR(10), sum INT)
{
   SELECT territory, '-all-', sum(sales)
   GROUP BY territory;
   SELECT '-all-', '-all-', sum(sales);
   SELECT '-all-', product, sum(sales)
   GROUP BY product;
}
```

Thus, redefinition can be used to support the GROUPING SET construct of DB2:

```
SELECT territory, product, sum(sales) FROM data
GROUP BY GROUPING SET ((territory), (product), ());
```

In AXL, GROUPING SET is supported through UDAs. The AXL compiler will generate `gssum`, to rewrite the GROUPING SET construct. Then, `gssum`, will be called as follows:

```
SELECT gssum(territory, product, sales)
FROM data;
```

The examples given so far illustrate the complete AXL language. Therefore, AXL manages to be quite powerful using a very small repertoire of new constructs beyond SQL.

4 New OLAP Functions

Constructs like ROLLUP and CUBE can be supported with techniques similar to those of Example 8, which only replicates the standard processing used by O-R systems for grouping sets. However, many special OLAP algorithms have been proposed for data with particular characteristics; these can be expressed using UDAs, while most DBMSs rely on a general algorithm. Therefore, UDAs can deliver much better performance, as illustrated by Bayesian classifiers. Unlike other data mining functions that are very difficult to support in SQL, these classifiers can be expressed well in SQL using supergroups. But the standard implementation of supergroups provided in DB2 yield much worse performance than that obtained using SQLAG UDAs[14].

Powerful aggregate extensions based on modifications and generalization of group-by constructs have recently been proposed by researchers, OLAP vendors, and standard committees. We will now show that this new functionality can be expressed in AXL, which also provides a good mechanism to simplify the implementation of these new aggregate constructs.

A new class of aggregate function, called OLAP function was introduced in [18]. An OLAP function is defined using a *window*. Windows extend the group-by construct by supporting multiple partitions, with ordering and row groups for each partition. The following example shows the use of multiple windows in a SQL query.

A business sells many products in many territories. It is desired to see the percentage contribution to last year's total sales of the sales from each combination of product and territory. In the same report, show the percentage contribution of each combination to its product's total sales across all territories, and the percentage contribution of each combination to its territory's total sales across all products.

EXAMPLE 9. *Percentage sales within whole, within product, and within territory*

```
SELECT   d.product, d.territory,
         d.sales/sum(d.sales) OVER Whole,
         d.sales/sum(d.sales) OVER By_prdct,
         d.sales/sum(d.sales) OVER By_tertry
FROM     data AS d
WHERE    d.year = 1999
WINDOW   Whole AS (),
         By_prdct AS (PARTITION BY d.product),
         By_tertry AS (PARTITION BY d.territory);
```

AXL supports this query by converting it into the following equivalent query that uses UDAs defined in Example 10.

```
SELECT mwin(territory, product, sales) FROM data;
```

EXAMPLE 10. *AXL implementation for a multi-window aggregate*

```
AGGREGATE mwin(territory CHAR(10),
               product CHAR(10), sales INT):
   (territory CHAR(10), product CHAR(10),
         wsum REAL, psum REAL, tsum REAL)
{
   TABLE summary (territory CHAR(10),
           product CHAR(10), sum INT) AS
     (SELECT gssum(territory, product, sales)
                FROM data);
   INITIALIZE: ITERATE: {
      INSERT INTO RETURN
      SELECT territory, product,
            sales/w.sum, sales/p.sum, sales/t.sum
      FROM      summary AS w p t
      WHERE     w.product = '-all-'
         AND    w.territory = '-all-'
         AND    p.product = product
         AND    p.territory = '-all-'
         AND    t.product = '-all-'
         AND    t.territory = territory
   }
}
```

The definition of UDA `mwin` uses a builtin aggregate of AXL: `assemble`. Given a stream of `values`, `assemble(value, n)` juxtaposes `n` values into one record.

5 Decision Tree Classifiers

The first step for most decision tree classifiers is to convert the training set into column/value pairs. This conversion, although conceptually simple, is hard to express succinctly in SQL. Given a training set, we want to convert it into a 4-column stream (RecId, Col, Value, YorN), meaning that the Col-th column of the RecId-th tuple has value Value and class label YorN.

For instance, for the usual PlayTennis table

Outlook	Temp	Humidity	Wind	PlayTennis
Sunny	Hot	High	Strong	No
Sunny	Hot	High	Weak	Yes
	...	...	... ...	

we want to derive the following table:

RecId	Col	Value	YorN
1	1	Sunny	No
1	2	Hot	No
1	3	High	No
1	4	Strong	No
2	1	Sunny	Yes
2	2	Hot	Yes
2	3	High	Yes
2	4	Weak	Yes
3	1	...	...
...	...	...	...

In AXL, we can define a new aggregate `dissemble` to do the conversion.

EXAMPLE 11. *Recasting into column/value pairs.*

```
AGGREGATE dissemble(v1 INT, v2 INT, v3 INT,
                    v4 INT, yorn INT):
   (col INT, val INT, YorN INT)
{
   INITIALIZE: ITERATE: {
      INSERT INTO RETURN
         VALUES  (1,v1,yorn), (2,v2,yorn),
                 (3,v3,yorn), (4,v4,yorn);
   }
}
```

In the query listed below, we dissemble a training set, tennis, which has 4 categorical columns, Outlook, Temp, Humidity, and Wind, and a class label column PlayTennis (the mcount aggregate is used here to generate a record id for each tuple in the tennis table):

```
INSERT INTO source
    SELECT mcount(),
            dissemble(Outlook, Temp, Humidity,
                        Wind, PlayTennis)
    FROM tennis;
```

The SPRINT Algorithm in AXL

The SPRINT algorithm proposed in [12] is implemented in AXL as shown in Example 12. For the `sprint` aggregate, the `source` table previously described is defined with the following `KEY` declaration to ensure that the records are sorted.

```
TABLE source ( RecId INT, Col INT, Val REAL,
                YorN INT, KEY (Col, Val));
```

Then, during the INITIALIZE and ITERATE step, we append each column/value pair into the `treenodes` table, which is sorted on `Value` for each `Col`. Then, we compute the current gini index (line 13) using UDF `giniudf`, which returns a new gini index if it is less than the gini index found at the previous best splitting position.

After we made one pass through the data, we pick the column that has the least gini value (line 17-19). The UDA `minpointvalue` is the same aggregate as the `minpoint` defined in Example 1, except that it also returns the splitting position and minimum gini, which will be used as the stop condition for the recursion on line 26. (If gini equals 0, then all the records in the current node have the same class label and no further classification is necessary.) The current split is recorded into a global table `result`. Next, we distribute each column/value pair in the `treenodes` table into its corresponding subnode. The classification for the next round starts by calling the `sprint` aggregate recursively on line 27.

A Categorical Decision Tree Classifier

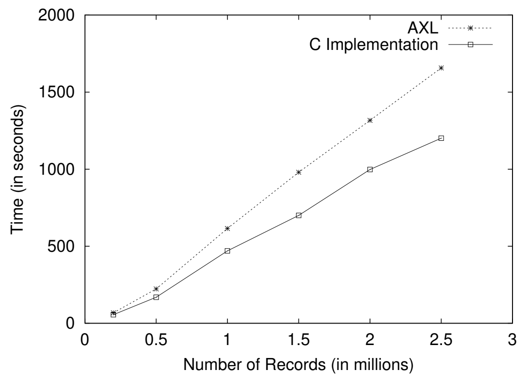
Here we present a decision tree classifer in AXL for training set consisting only categorical attributes. The complete classifier algorithm is shown in Example 13.

First, we insert the current record into the `treenodes` table. Then we update the class histogram kept in the `summary` table for each column/value pair. If we have not met the column/value pair before, we will insert it on line 14. Here, SQLCODE is a reserved word of AXL, and it keeps the return status of the previous SQL statement. If SQLCODE is 0, then the previous UPDATE statement is unsuccessful and a new record is to be inserted.

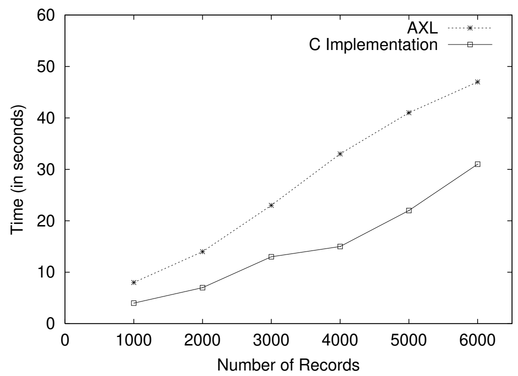
The TERMINATE step will first compute the gini index for each column using the histogram we accumulated during the INITIALIZE/ITERATE steps. Then, on line 24, we select the splitting column that has the minimal gini index. A new sub-branch of the decision tree is generated for each value in the column. After recording the current split into the `result` table, we call the classifier recursively to further classify the sub nodes. The GROUP-BY clause on line 36 partitions the records in `treenodes` into at most MAXVALUE subnodes, where MAXVALUE is the maximum total number of different values a column has.

Using the `classify` aggregate, classification on the `PlayTennis` training set (see Section 5) can be solved using the following query:

```
SELECT classify(t.RecId, 0, t.Col, t.Val, t.YorN)
FROM ( SELECT mcount() RecId,
            dissemble(Outlook,Temp,Humidity,
            Wind,PlayTennis) AS (Col, Val, YorN)
       FROM tennis) AS t;
```



(a) SPRINT Algorithm (Function 7)



(b) The German Credit Problem

Figure 1: Performance Comparison of AXL and C

6 Implementation of AXL

The AXL compiler translates AXL programs into C++ code. The classifier algorithm in Example 12, for instance, is compiled into more than 2100 C++ code. AXL adopts an open interface for its physical data model, so that the system can link with a variety of physical database implementations. Currently, we use the Berkeley DB library[19] as our main storage manager.

AXL supports both persistent tables and temporary tables. Temporary tables are declared as local variables in the program and are memory based. Aggregates in AXL are hash-based by default. However, we also allow the use of predicates like SORT BY column or SORT BY

Example 12. *The SPRINT Algorithm in AXL*

```
AGGREGATE sprint(iNode INT, RecId INT, iCol INT, iValue REAL, iYorN INT)
{
  TABLE treenodes(RecId INT, Col INT, Value REAL, YorN INT, KEY(Col, Value));
  TABLE summary(Col INT, SplitGini REAL, SplitVal REAL, Yc INT, Nc INT);
  TABLE split(RecId INT, LeftOrRight INT, KEY (RecId));
  TABLE mincol(Col INT, Value REAL, Gini REAL);
  TABLE node(Node INT) AS VALUES(iNode);

  INITIALIZE : ITERATE : {
        INSERT INTO treenodes VALUES (RecId, iCol, iValue, iYorN);
        UPDATE summary
              SET Yc=Yc+iYorN, Nc=Nc+1-iYorN,
                 (SplitGini, SplitVal) = giniudf(Yc, Nc, N, SplitGini, SplitVal)
              WHERE Col = iCol;
  }
  TERMINATE : {
        INSERT INTO mincol
              SELECT minpointvalue(Col, SplitGini, SplitVal)
              FROM summary;
        INSERT INTO result
              SELECT n.Node, m.Col, m.Value
              FROM mincol m, node n;
        INSERT INTO split
              SELECT t.RecId, (t.Value>m.Value)
              FROM treenodes t, mincol m
              WHERE t.Col=m.Col AND m.Gini > 0;
        SELECT sprint(n.Node*2+s.LeftOrRight, t.RecId, t.Col, t.Value, t.YorN)
        FROM treenodes AS t, split AS s, node AS n
        WHERE t.RecId = s.RecId
        GROUP BY s.LeftOrRight;
  }
}
```

GROUPBY in the redefinition of UDAs; this induces a sort-based computation for the aggregates that follow and use the same group-by columns.

The runtime model of AXL is based on data pipelining. In particular, all UDAs, including recursive UDAs that call themselves, are pipelined, which means tuples inserted into the RETURN relation during the INITIALIZE/ITERATE steps are returned to their caller immediately. In order to do this, all local variables (temporary tables) declared inside the body of a UDA are assembled into a `state` structure which will be passed into the UDA for each INITIALIZE, ITERATE, and TERMINATE calls, to assure that these internal data are saved between the calls.

We compared the performance of algorithms written in AXL with that written in C. In Figure 1(a) we show the performance of the SPRINT algorithm on the synthetic dataset Function 7 [20]. As shown in Figure 1(b), performance is somewhat worse for categorical classifier, where operations for which AXL is optimized, such as searching and scanning of tables, play a comparatively lesser role. This classifier was described more extensively in [23] and is summarized for convenience of the reader. The training set used is based on the "German credit" dataset, which is converted to 19 categorical attributes [7]. These results were obtained on a Pentium II 500 MHz machine running Linux 2.2.9.

7 Conclusions

In this paper, we have shown that user-defined aggregates expressed in AXL are effective with both advanced OLAP applications and data mining applications. Moreover, AXL is also effective with time series applications, which are now supported in O-R systems by special time-series datablades. Indeed, UDAs' stream-oriented computation model is very effective at handling sequences: double-bottom queries, and moving windows aggregates are easy to express in AXL [23].

In summary, AXL's UDAs can be effective on the whole spectrum of decision support applications, including advanced OLAP queries, time series analysis, and data mining functions. Currently we are testing the limits of AXL by developing various data mining functions, such as Apriori and clustering algorithms, and measuring their performance against the same functions coded in C++. This process also drives the addition of new optimization techniques to the AXL compiler. Our final objective is to achieve performance close to those achievable by coding in C; progress on this front will be reported at the workshop. Even at the present level of performance, AXL provides a desirable tool for the rapid prototyping of new decision support applications (delivery performance better than, say, Java or PL/SQL).

In the longer term, many techniques now used to execute SQL in distributed and parallel environments will be added to AXL, providing opportunities for distributed and parallel data mining.

Example 13. *Categorical Classifier Expressed in AXL*

```
AGGREGATE classify(RecId INT, iNode INT, iCol INT, iValue INT, iYorN INT)
{
   TABLE treenodes(RecId INT, Node INT, Col INT, Value INT, YorN INT);
   TABLE mincol(Col INT, MinGini REAL);
   TABLE summary(Col INT, Value INT, Yc INT, Nc INT, KEY {Col,Value});
   TABLE ginitable(Col INT, Gini REAL);

   INITIALIZE : ITERATE : {
         INSERT INTO treenodes
                  VALUES(RecId, iNode, iCol, iValue, iYorN);
         UPDATE summary
                  SET Yc=Yc+iYorN, Nc=Nc+1-iYorN
                  WHERE Col = iCol AND Value = iValue;
         INSERT INTO summary
                  SELECT iCol, iValue, iYorN, 1-iYorN
                  WHERE SQLCODE=0;
   }
   TERMINATE : {
         INSERT INTO ginitable
                  SELECT Col, sum((Yc*Nc)/(Yc+Nc))/sum(Yc+Nc)
                  FROM summary
                  GROUP BY Col;
         INSERT INTO mincol
                  SELECT minpointvalue(Col, Gini)
                  FROM ginitable;
         INSERT INTO result
                  SELECT iNode, Col
                  FROM mincol;
         SELECT classify(t.RecId, t.Node*MAXVALUE+m.Value+1, t.Col, t.Value, t.YorN)
         FROM treenodes AS t,
               ( SELECT tt.RecId RecId, tt.Value Value
                  FROM treenodes tt, mincol m
                  WHERE tt.Col=m.Col AND m.MinGini > 0
               ) AS m
         WHERE t.RecId = m.RecId
         GROUP BY m.Value;
   }
}
```

8 REFERENCES

[1] D., Chamberlin, "Using the new DB2, IBM's Object-Relational Database System," Morgan Kaufmann, 1996.

[2] R. Chandra and A. Segev. Managing Temporal Financial Data in an Extensible Database. In *Proc. of the 19th VLDB*, pages 302–313, 1993.

[3] Surajit Chaudhuri, Usama M. Fayyad, Jeff Bernhardt: Scalable Classification over SQL Databases. ICDE 1999: 470-479.

[4] D. Chatziantoniou and K. A. Ross, "Querying Multiple Features of Groups in Relational Databases."Proceedings of the 1996 VLDB Conference, September 1996.

[5] D. Chatziantoniou and K. A. Ross, "Groupwise Processing of Relational Queries." Proceedings of the 1997 VLDB Conference, pages 476-485, August 1997.

[6] Charles Elkan. "Boosting and Naive Bayesian Learning". Technical report no cs97-557, Dept. of Computer Science and Engineering, UCSD, September 1997.

[7] StatLog Datasets, http://www.ncc.up.pt/liacc/ML/statlog /datasets/german/german.doc.html

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

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

[10] ISO DBL LHR-004 and ANSI X3H2-95-364, "(ISO/ANSI Working Draft) Database language SQL3", Jim Melton (ed), dated 1995.

[11] S. Sarawagi, S. Thomas, R. Agrawal, "Integrating Association Rule Mining with Relational Database Systems: Alternatives and Implications". In *SIGMOD, 1998.*

[12] J. C. Shafer, R. Agrawal, M. Mehta, "SPRINT: A Scalable Parallel Classifier for Data Mining," In *VLDB 1996.*

[13] P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: Design and implementation of a sequence database system. submitted for publication, 1996.

[14] H. Wang and Carlo Zaniolo, "User-Defined Aggregates for Datamining," ACM SIGMOD Workshop Data Mining and Knowledge Discovery, Philadelphia, 1999.

[15] H. Wang, The SQL-AG System, http://magna.cs.ucla.edu/~hxwang /sqlag/sqlag.html

[16] Haixun Wang, The AXL System, http://vesuvio.cs.ucla.edu/~hxwang/axl

[17] Haixun Wang and Carlo Zaniolo, "User Defined Aggregates in Object-Relational Systems", in ICDE 2000, San Diego, California, USA.

[18] F. Zemke, K. Kulkarni, A. Witkowski, B. Lyle, "Introduction to OLAP functions" ISO/IEC JTC1/SC32 WG3: YGJ-068 = ANSI NCITS H2-99-154r2.

[19] Sleepycat Software, "The Berkeley Database", http://www.sleepycat.com.

[20] R. Agrawal, T. Imielinski, and A. Swami. "Database Mining: A Performance Perspective." IEEE Trans. on Knowledge and Data Engineering, 5(6), Dec. 1993.

[21] Haixun Wang and Carlo Zaniolo, "User Defined Aggregates in Database Languages", 7th Int. Workshop on Database Programming Languages, Kinloch Rannoch, Scotland, 1999.

[22] I. Motakis and C. Zaniolo, "Temporal Aggregation in Active Database Rules". In *SIGMOD'97*.

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

[24] T. Imielinks Tomasz, Aashu Virmani, Amin Abdulghani, "DataMine: Application Programming Interface and Query Language for Database Mining", KDD 1996: 256-262.

[25] Rosa Meo, Giuseppe Psaila, Stefano Ceri, "A New SQL-like Operator for Mining Association Rules", VLDB 1996: 122-133.