Using SQL to Build New Aggregates and Extenders for Object-Relational Systems

Haixun Wang   Carlo Zaniolo

Computer Science Department
University of California at Los Angeles†

Abstract

User-defined Aggregates (UDAs) provide a versatile mechanism for extending the power and applicability of Object-Relational Databases (O-R DBs). In this paper, we describe the AXL system that supports an SQL-based language for introducing new UDAs. AXL is easy to learn and use for database programmers because it preserves the constructs, programming paradigm and data types of SQL (whereas there is an 'impedance mismatch' between SQL and the procedural languages of user-defined functions currently used in O-R DBs). AXL will also inherit the benefits of database query languages, such as scalability, data independence and parallelizability. In this paper, we show that, while adding only minimal extensions to SQL, AXL is very powerful and capable of expressing complex algorithms efficiently. We demonstrate this by coding data mining functions and other advanced applications that, previously, had been a major problem for SQL databases.

Due to its flexibility, SQL-compatibility and ease of use, the AXL approach offers a better extensibility mechanism, in several application domains, than the function libraries now offered by commercial O-R DBs under names such as Datablades or DB-Extenders.

† Los Angeles CA 90095, `hxwang|zaniolo@cs.ucla.edu`

Proceedings of the 26th VLDB Conference, Cairo, Egypt, 2000.

1 Introduction

The explosive growth of new database applications has, in several cases, outpaced the albeit dramatic progress made by database technology. In fact, the great success of new application areas often serves as a grim reminder of the limitations from which DBMSs still suffer in terms of power and extensibility. For instance, the newly introduced Object-Relational (O-R) systems offer great improvements in generality, extensibility, and query power; yet O-R systems do not support well data mining applications—a leading growth area for data-intensive applications. Problems also occur in many other application areas, where datablades and similar function libraries need better flexibility and integration with SQL. In this paper, we show that many of these problems can be solved, or ameliorated, by user-defined aggregates (UDAs), which often provide a more flexible and powerful mechanism for extending DBMSs than user-defined functions (UDFs) used today in this role. For instance, an aggregate can take as argument the whole SQL relation rather than fields in individual tuples, as UDFs do. Unfortunately, UDAs are less understood and developed from a technological viewpoint than UDFs, which have greatly benefitted from the ADT advances made in programming languages, while UDAs are primarily a DB-centric concept and have received less attention. In fact, while UDAs were part of Postgres [18] and early SQL3 [11, 10] proposals, and also supported in Informix [12], they have been left out from the recently released SQL-99 specifications.

Therefore, as today, UDAs remain a topic rich with research challenges and opportunities, since they find many important application areas, such as DB-centric data mining. At UCLA, we developed the SQL-AG system that supports UDAs on top of DB2; besides implementing the original SQL3 specifications for UDAs, SQL-AG extends them to support new forms of aggregation, such as online aggregation, through a mechanism called *early returns* [23]. Aggregates only producing early returns are monotonic with respect to set-containment, and can, therefore, be used in recursive SQL queries with no restriction or modification to the current systems [23]. Monotonic aggregates support efficiently Bill of Materials applications, transitive closures with greedy optimization, and other complex queries that had been a problem for relational query languages since their introduction [23].

In the original SQL3 proposal, new UDAs are actually encoded through three UDFs that, respectively, define the computation to be performed (i) for the first value in the stream, (ii) for each successive value, and (iii) at the end of the stream. Unfortunately, programming UDFs for O-R systems using procedural languages can be exceedingly difficult even for knowledgeable programmers [15]. The difficulty of writing and debugging UDFs is compounded by the fact that, to achieve reasonable performance, these UDFs will normally execute 'unfenced' [2] in the same address space as the database system—thus the use of efficient procedural languages, such as C, could compromise the safety of the system. Clearly, UDAs defined via multiple procedural language UDFs, as suggested in the original SQL3 proposal, will suffer from similar usability problems.

Our approach to solve these problems consists in providing a high-level language for defining new aggregates. Since all users are already familiar with SQL, we will strive to design a language as close as possible to SQL; this will make the new language easier to learn and use, avoid the many problems connected with the introduction of a new language, and eliminate the risk of 'impedance mismatch' in data types and programming styles that is bound to occur if UDAs are written in any other language. In addition, this approach inherits the many advantages of databases and their query languages, including scalability, data independence, and parallelizability.

The main challenge facing this approach was the limited expressive power of SQL, which made us wonder if our objective was achievable at all. Our Simple Aggregate Definition Language (SADL) project [22] represented an important experiment to test the limits and feasibility of our SQL-centric approach. SADL is a 'barebone' language, which only supports basic SELECT statements. Even so, SADL was sufficient to express some aggregates, boosting our confidence that our ultimate goal was achievable. On the other hand, as we explored the issues of performance, scalability, and expressive power required for advanced applications, SADL showed many problems. One is the fact that SADL kept the UDA structure proposed in the original SQL3 specifications, where the definition of a new aggregate is broken down into several functions. This structure leads to poorly structured programs and inefficiency. Another problem with SADL is that it does not support the definition and use of auxiliary tables, nor does it support updates on tables. Therefore, a new language called AXL (for Aggregate Extension Language) was designed to solve all these problems; the AXL system also uses an architecture very different from SADL. For instance, while SADL implementation uses query interpretation, and relies on in-memory tables, AXL relies on compilation and makes full use of secondary memory tables.

This paper is organized as follows. In the next section, we describe our systems SQL-AG and SADL previously developed at UCLA to deal with UDAs and their limitations. Then in Sections 3, 4, 5 and 6 we describe AXL in various application domains. In section 7 we describe its implementation. In section 8 we discuss opportunities for future research.

2 The SQL-AG System and SADL

At UCLA, we have developed the SQL-AG [20] system that supports and extends the UDA specifications originally proposed for SQL3 [11]. For instance, we can define the standard **avg** aggregate as shown in Example 1.

Example 1 *Defining the standard avg aggregate*

```
AGGREGATE myavg(INT)
 RETURNS REAL
 STATE state_type
 INITIALIZE avg_single
 ITERATE avg_multi
 TERMINATE avg_terminate

typedef struct state_s {
  long sum; long cnt;
} state_type;
void avg_single(long *value, state_type *s)
{ s→sum = *value;
  s→cnt = 1;
}
void avg_multi(long *value, state_type *s)
{ s→sum += *value;
  s→cnt++;
}
void avg_terminate(state *s, float *result)
{ *result = s→sum/s→cnt;
}
```

Basically, 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 value in the stream, for each successive value, and when the end of stream is detected and the final value of the aggregate must be returned. Thus, in our example, the C procedure *avg_single* initializes the current sum to the first value and the current count to 1, *avg_multi* adds the new value to sum and increases count by 1, and *avg_terminate* returns the final average from *state*.

It is important to observe that while traditional SQL2 aggregates are (dependent on repetitions, but) independent of the order in which the computation streams through data, UDAs as defined in the original SQL3 specifications and implemented in SQL-AG can depend on such order. Recent SQL extension for aggregates, supporting partition and windows for OLAP applications [25], also relies on the the order of data. Indeed in many applications, such as cumulative aggregates and moving-windows aggregates used in time-series [14], the fact that the data is sorted by their time stamps is part of the application logic. On the other hand, a direct application of online aggregates on stored data, normally relies on the fact that the data is not skewed [8]. Thus, aggregates that are most useful in advanced applications are often designed to take full advantage on the particular properties of the data.

Two versions of SQL-AG were implemented, the first on Oracle, using PL/SQL, and the second for IBM DB2. Here we describe this second version, which is significantly more powerful and efficient than the other. DB2 supports user-defined functions (UDFs) but not user-defined aggregates. The SQL-AG system supports SQL queries with UDAs by transforming them into DB2 queries that use scratch-pad UDFs to emulate the functionality of the corresponding UDAs [2]. For instance, moving average of stock price in the following query:

```
SELECT company, myavg(price)
FROM stock-closing
GROUP BY company
```

is translated by SQL-AG into the query which can be executed by DB2:

```
SELECT company, myavg_out(company)
FROM stock-closing
WHERE myavg_groupby(company,price)=1
GROUP BY company
```

Each UDA, named say agg, is implemented with two automatically generated DB2 UDFs, namely, agg_groupby and agg_out. The UDF agg_groupby performs the actual computation and it is applied to every record for aggregation. It uses a hash table to keep the aggregate value of each group. The UDF agg_out(group) retrieves from the hash table the last value computed by agg_groupby for group. A detailed description of SQL rewriting can be found in [20].

A key improvement made by the SQL-AG system developed at UCLA [20] 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 this paper.

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, *avg_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(INT)
RETURNS REAL
STATE state_type
INITIALIZE avg_single
ITERATE avg_multi
PRODUCE avg_produce
TERMINATE avg_terminate


int avg_produce(state *s, float *result)
{ if (s→cnt % 100 == 0) {
      *result = s→sum / s→cnt;
      return 1;
  } else
      return 0;
}
```

This version of myavg contains both early returns and final returns. The rules required to map UDAs with both early returns and final returns into equivalent SQL queries with scratchpad UDFs are more complex, as described in [20].

Monotonic Aggregation. An interesting special case is when the aggregate only contains early returns, i.e., the TERMINATE function avg_terminate 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 [23, 22].

Performance. We compared the performance of native DB2 builtins against SQL-AG UDAs on a Ultra SPARC 2 with 128 megabytes memory and reimplemented the SQL built-ins as UDAs. As discussed in [22], when aggregation contains no group-by columns, there is a slight performance penalty resulted from calling UDFs. There are actually situations where the fact that a hash-based aggregation is used instead of a sort-based one yielding better performance than built in aggregates. Thus UDAs can be implemented with nearly the same performance as the standard built-ins.

SADL. UDAs defined using a procedural languages such as C suffer from the same problems as the C-defined user defined functions[15]; these problems include the difficulty of developing and debugging applications, and a loss of optimizability and parallelizability. An obvious solution to these problems consists in providing a high-level language for the definition of aggregates. In [22] we presented a simple aggregate definition language (SADL), which was designed to have SQL-like syntax, data types, and semantics, for ease of learning and use. In SADL, the user codes the INITIALIZE function and the other functions required to introduce a new aggregate using SELECT statements.

While simple aggregates can be expressed easily, more complex ones could not be expressed in SADL, which lacks the ability of introducing temporary tables and calling UDAs recursively. This led to the design of a new language, named AXL, where new tables and aggregates can be defined as part of the definition of an aggregate, and updates on tables are supported along with the recursive invocation of aggregates.

3 AXL

We now introduce AXL by examples; a more complete discussion can be found in [21]. 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 labelled respectively INITIALIZE and ITERATE and TERMINATE. Thus, INITIALIZE inserts the value taken from the input stream and sets the count to 1. The ITERATE statements update 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 it to RETURN; for conformity with SQL, 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 ;
   }
}
```

We now define a `minpoint` aggregate that returns the point where a minimum occurs rather than the value of the minimum.

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;
   }
}
```

Monotonic Count. Another interesting aggregate is the monotonic count mcount that returns the running count for each new input value. This aggregate is monotonic with respect set containment. Indeed for a set of cardinality 3 it returns $\{1, 2, 3\}$, and once a new element is added to this set it returns $\{1, 2, 3, 4\}$. Observe that this second set is a superset of the first; the traditional count would instead return singleton sets $\{3\}$ and $\{4\}$, where the second is not a superset of the first. The monotonic aggregate mcount can be used freely in recursive SQL queries to express many computations that would be hard or inefficient to express otherwise; several examples are given in [22]. Here we will later use it to assign a sequence number to the tuples of a relation.

Example 5 *Monotonic Count*

```
AGGREGATE mcount(Next INT) : INT
{
   TABLE state(cnt INT);
   INITIALIZE : {
      INSERT INTO state VALUES(1);
      INSERT INTO RETURN VALUES (1);
   }
   ITERATE : {
      UPDATE state SET cnt=cnt+1;
      INSERT INTO RETURN
         SELECT cnt FROM state;
   }
 }
```

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 6 defines a recursive aggregate `alldesc` to find all the descendants of a given person.

Example 6 *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 descendents of Tom.

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

AXL also supports a redefinition construct, and SQLCODE construct which are described in the next sections. But, basically, 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. In the next section, we show how AXL can be used to support OLAPs and other powerful new forms of aggregation; in the section that follows we use AXL to support some data mining functions.

4 Data Mining in AXL

As a first example of the many uses of AXL, consider the data mining methods used for classification. Say for instance, that we want to classify the value of Play as a 'Yes' or a 'No' given a training set such as that shown in Table 1.

Outlook	Temp	Humidity	Wind	Play
Sunny	Hot	High	Weak	No
Sunny	Hot	High	Strong	No
Overcast	Hot	High	Weak	Yes
Rain	Mild	High	Weak	Yes
Rain	Cool	Normal	Weak	Yes
Rain	Cool	Normal	Strong	Yes
Overcast	Cool	Normal	Strong	No
Sunny	Mild	High	Weak	No
Sunny	Cool	Normal	Weak	Yes
Rain	Mild	Normal	Weak	Yes
Sunny	Mild	Normal	Strong	Yes
Overcast	Mild	High	Strong	Yes
Overcast	Hot	Normal	Weak	Yes
Rain	Mild	High	Strong	No

Table 1: The relation **PlayTennis**

We now describe the implementation in AXL of categorical decision tree classifiers. Bayesian classifiers will be discussed in the next section.

Decision Tree Classifier. 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. Consider the **PlayTennis** relation as shown in Table 1. We want to convert it into a new stream of 4 columns **(RecId, Col, Value, YorN)**, meaning that the **Col**-th column of the **RecId**-th tuple in relation **PlayTennis** has value **Value** and its **Play** column has value **YorN**.

In AXL, we can define a new aggregate `dissemble` to solve the problem.

Example 7 *Dissemble a relation 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);
   }
}
```

Following is the query to dissemble the `PlayTennis` table (the `mcount` aggregate is used here to generate a record id for each tuple in the tennis table):

```
SELECT mcount(1),
 dissemble(Outlook, Temp, Humidity, Wind, Play)
FROM PlayTennis;
```

The complete classifier algorithm is shown in Example 8. The INITIALIZE and ITERATE steps share the same block of code. 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 17. 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 23, 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 used here, `minpointvalue`, is the same aggregate as the `minpoint` defined in Example 4 except that it also returns the minimum value, which will be used as the stop condition for the recursion on line 37. (If gini equals 0, then all the records in the current node has the same class label and no further classification is necessary.) 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 40 partitions the records in `treenodes` into MAXVALUE subnodes, where MAXVALUE is the largest number of different values in any of the table columns (three for Table 1).

Using the `classify` aggregate, classification on the tennis table 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,Play)
       AS (Col, Val, YorN)
    FROM PlayTennis) AS t;
```

Example 8 *Using Recursive Aggregates to Implement a Classifier 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);
   TABLE summary(Col INT, Value INT, Yc INT, Nc INT,
                 KEY {Col,Value});
   TABLE ginitable(Col INT, Gini INT);

   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 AS tt, mincol AS m
         WHERE tt.Col=m.Col AND m.MiniGini > 0
      ) AS m
      WHERE t.RecId = m.RecId
      GROUP BY m.Value;
   }
}
```

5 Group-By Modifiers

Powerful aggregate extensions based on modifications and generalizations of group-by constructs have recently been proposed by researchers, OLAP vendors, and standard committees [5, 25]. Here, we show how these aggregate extensions can also be expressed in AXL, as an alternative and more flexible mechanism to achieve their advanced functionality.

Consider the following query to an `employee` relation:

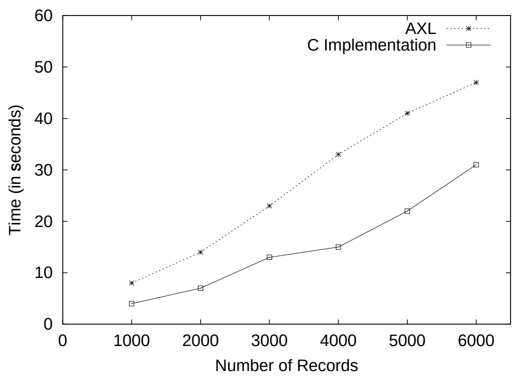

Figure 1: Performance Comparison of AXL and C

For each division, show the average salary of senior managers who make 3 times more than the average employees, and the average salary of senior engineers who make 2 times more than the average employees (in the same output record).

All aggregates in this query are grouped by the same attribute (i.e., division). To express this query in SQL2, we need to use joins and subqueries, and then three passes through the `employee` relation will be needed to produce the answer. To solve these problems, a new construct called SUCH THAT was proposed in [5]. Example 9 shows the use of this construct to express our query. Therefore, this extension uses variables, such as X and Y, that range over groups and are qualified by the SUCH THAT conditions.

Example 9 *SQL extension SUCH THAT proposed in [5]*

```
SELECT division, avg(X.salary), avg(Y.salary)
FROM employee
GROUP BY division : X, Y
SUCH THAT X.title= 'senior manager' AND
          X.salary > 3 * avg(salary) AND
          Y.title= 'senior engineer' AND
          Y.salary > 2 * avg(salary))
```

Queries with SUCH THAT constructs can be implemented in AXL by mapping the queries into UDAs according to simple transformation rules.

Example 10 *Rewriting the SUCH THAT construct using a UDA*

```
AGGREGATE mscan2(title CHAR(20), salary INT,
                 querytitle CHAR(20), INT ratio) : 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 = querytitle;
   }
   TERMINATE: {
      SELECT avg(fsalary) FROM finalstate
      WHERE fsalary >
      (SELECT ratio*sum/cnt FROM allstate);
    }
}
```

Then, using aggregate `mscan2` defined above, we rewrite Example 11 into the following query:

```
SELECT  division,
        mscan2(title, salary, 'senior manager', 3),
        mscan2(title, salary, 'senior engineer', 2)
FROM employee
GROUP BY division;
```

We next discuss the implementation of a naive Bayesian classifier, to illustrate both the power and the limitations of OLAP extensions recently introduced in SQL.

Bayesian Classifiers. The Boosted Bayesian Classifier [7] was the winner of the KDD'97 data mining competition. Its derivation algorithm consists of a main phase that produces a *Naive Bayesian* classifiers and of a boosting phase, that normally produces a (modest) performance improvement.

The Naive Bayesian classifier makes probability-based predictions as follows. Let A_1, A_2, …, A_k be attributes, with discrete values, used to predict a discrete class C. (For the example at hand, we have four prediction attributes, $k = 4$, and C = 'Play'.) For attribute values a_1 through a_k, the optimal prediction is the value c for which $Pr(C = c|A_1 = a_1 \wedge \ldots \wedge A_k = a_k)$ is maximal. By Bayes' rule, and assuming independence of the attributes, this means to classify a new tuple to the value of c that maximize the product of $Pr(C = c)$ with:

$$\prod_{j=1,\ldots,K} Pr(A_j = a_j|C = c)$$

But these probabilities can be estimated from the training set as follows:

$$Pr(A_j = a_j|C = c) = \frac{count(A_j = a_j \wedge C = c)}{count(C = c)}$$

To compute the numerators and denominators in the above formula, we can use the GROUPING SET construct as shown in Example 11. It's AXL UDA equivalent is shown in Example 12.

Example 11 *Using DB2's GROUPING SET*

```
SELECT Outlook, Temp, Humidity, Wind,
       Play, count(*)
FROM PlayTennis
GROUP BY GROUPING SETS ((Outlook, Play),
   (Temp, Play), (Humidity, Play),
   (Wind,Play), (Play));
```

Example 12 *Using a UDA defined in AXL*

```
AGGREGATE assemble(A1 INT, A2 INT, A3 INT,
                   A4 INT, C INT) :
               (col CHAR(20), class INT)
{
   INITIALIZE: ITERATE: {
      INSERT INTO RETURN VALUES
         (A1, C), (A2, C), (A3, C),
         (A4, C), ('-all-', C);
   }
}
SELECT t.col, t.class, count(*)
FROM (SELECT assemble(Outlook, Temp,
                Humidity, Wind, Play)
        FROM PlayTennis) AS t
GROUP BY t.col, t.class;
```

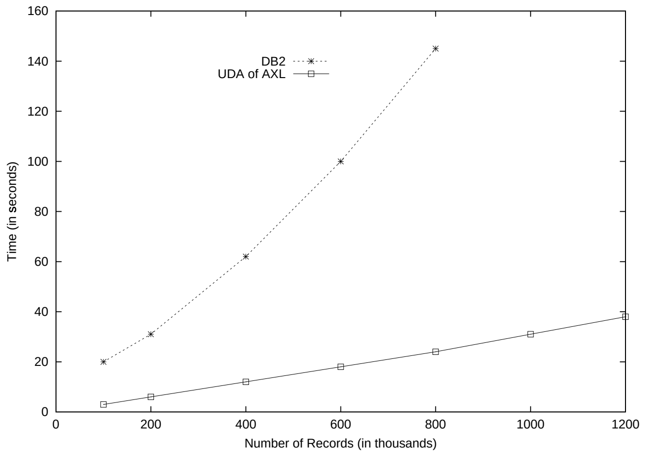

Figure 2: Bayesian Classifiers: grouping sets vs. UDAs

All counts needed in Example 12 are computed in one pass through the data using the hash-based method that is AXL's default approach, while in commercial systems, GROUPING SETS are often implemented as a cascade of sorting operations. As illustrated by Figure 2, AXL's specialized UDA yields a substantial speed-up, and improved scalability (DB2 on our workstation refused to handle more than 800,000 records generated as a synthetic data set).

Therefore, while OLAP extensions recently added to SQL extend its expressive power, their implementation cannot be expected to be optimal for all possible situations. There are many situations, where algorithms other than those used by the database vendor are better for the particular dataset at hand. In the case of categorical classifiers, for instance, the number of possible values is normally such that a hash-based aggregation is applicable and produces the best results.

More powerful OLAP functions are now being added to SQL [25]. But since their implementation can only be optimized for 'typical' situations, there will remain a need for UDAs to serve special situations, and for a high-level language to facilitate their implementation.

6 Datablades and Time Series

Databases are now pulled in different directions by the need to keep the SQL standards while supporting new data types and advanced applications. This has given birth to a proliferation of vendors' packages for new data types and application areas, such as, text, images, video, audio, time-series, spatial data, and others. These packages include IBM's DB2 Extenders, Informix' Data Blades, Oracle's Data Cartridges, and Sybase's Snap-Ins, which are basically libraries of functions that can be called from an SQL query on attribute values and blobs. While this approach is adequate for certain applications, it suffers from several limitations. For instance, the functions take as operands individual tuples rather than tables, and the canned operators provided by a datablade package cannot be extended easily. The first limitation is particularly obvious in data mining applications, which are devoted to finding interesting statistical correlations in the relation. We believe that aggregate-based extenders provide a better basis for data-mining datablades; in fact, AXL provides a powerful facility for writing new data mining functions, or for combining and extending existing ones to match the needs of the application at hand.

An area where the limitations of datablades have long been recognized is that of time-series for which researchers have proposed extensions that are more flexible and better integrated with SQL [17] than commercial time-series datablades. Indeed, a sequence of time-value pairs can naturally be viewed as a relation with ordered tuples, and time-series queries can be formulated via SQL-like query languages [3, 17]. Moreover, it is hard to imagine a more natural model for sequences and ordered relations than the stream model already used by our UDAs. The following examples illustrate the use of temporal UDAs on data that is stored and processed according to their time stamps. Also, we have made much use of UDAs in TEnORs (Temporally Enhanced OR System), to support a valid time extension of SQL [9].

TEMPORAL EXTENDERS. *We have a sequence of events, each of which is active during a certain interval (from, to). Find out at which point of time we have the largest number of active events.*

Example 13 *Active Intervals*

```
AGGREGATE density(from TIME, to TIME)
      : (time TIME, count INT)
{   TABLE state(time TIME, count INT) AS (0,0);
    TABLE active(endpoint TIME);
    INITIALIZE: ITERATE: {
       DELETE FROM active WHERE endpoint < from;
       INSERT INTO active VALUES(to);
       UPDATE state
          SET time=from, count=count+1
          WHERE count < (SELECT count(*) FROM active);
    }
    TERMINATE: {
       INSERT INTO RETURN
          SELECT time, count FROM state;
    }
}
```

Under the assumption that events are sorted by increasing time, we can use the following query to find the point of time that has the largest number of active events.

Example 14 *Coalescing*

```
SELECT density(from,to) FROM events;
```

When events are not sorted by time, we can create a new aggregate `sortdensity` that wraps around the aggregate defined in Example 13. In aggregate `sortdensity`, we specify the ordering of the input stream. Then, in Example 14, we use `sortdensity` instead of `density` to find the point of time that has largest number of active events on an unordered stream.

Example 15 *Ordering*

```
AGGREGATE sortdensity(from TIME, to TIME)
    : (time TIME, count INT)
{
    ORDER BY from ASC;
    INSERT INTO RETURN
       SELECT density(from, to);
}
```

This example, illustrates AXL's *redefinition* facility which takes an input stream and transforms it using previously defined aggregates and sorting.

Let us consider now the well-known problem of coalescing after projection in a temporal table. We define the aggregate `coalesce`, which takes two parameters: `from` is the start time, `to` is the end time. Under the assumption that tuples are sorted by increasing start time, then we can perform the task in one scan of the data. In the ITERATE routine, when the new interval overlaps the current interval kept in the `state` table, we coalesce the two intervals into one interval that ends with the larger of the end time. Otherwise, the current interval is returned while the new interval becomes the current one.

Example 16 *Coalescing*

```
AGGREGATE coalesce(from TIME, to TIME)
        : (start TIME, end TIME)
{   TABLE state(cFrom TIME, cTo TIME);
    INITIALIZE: {
       INSERT INTO state VALUES(from,to);
    }
    ITERATE : {
        UPDATE state SET cTo = to
            WHERE cTo >= from AND cTo < to;
        INSERT INTO RETURN
            SELECT cFrom, cTo FROM state
            WHERE cTo < from;
        UPDATE state
            SET cFrom = from, cTo = to
            WHERE cTo < from;
    }
    TERMINATE: {
        INSERT INTO RETURN
            SELECT cFrom, cTo FROM state;
    }
}
```

7 Implementation of AXL

The AXL compiler translates AXL programs into C++ code. The classifier algorithm in Example 8, 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[26] 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 GROUPBY in a UDA to force sort-based aggregation.

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 is passed into the UDA for each INITIALIZE/ITERATE/TERMINATE calls

All the constructs described in this paper, but redefinition facility described in Example 15, are functional in the current AXL prototype that contains more than 33,000 lines of C++ code. We are now adding more SQL data types, O-R database extensions and a richer set of supporting indexes and storage structures. We expect the complete and more robust system will eventually have 90,000 lines of code.

AXL UDAs can either be used as stand-alone programs or, imported into DB2 using the SQL-AG approach (with limitations due to the fact that we use UDFs that return a single value for each call).

8 Conclusions

The goal of extending database systems to support new applications has been the focus of much interest and activity for database researchers and commercial vendors. While the previous efforts have concentrated on UDFs and ADTs, we have shown here that UDAs can play a major role in supporting new applications—including datamining applications which have proved a real challenge for O-R DBs [15]. Since aggregates can be viewed as query operators on tables, they can be defined using SQL (while UDFs are basically operators on tuples). The SQL statements defining a UDA can in turn call other UDAs (often using recursion) to yield great expressive power and flexibility. The AXL prototype has turned this simple ideas into an efficient implementation that builds on the lessons learned from our two previous UDA prototypes—i.e., the SQL-AG system [23] and SADL [22].

Using AXL, the database programmer can extend the functionality of the database system using the familiar programming style and data types of SQL. While ease of use is not a small advantage (particularly given the difficulty of extending the database system through UDFs), many other gains are to be expected. Indeed, the traditional benefits of SQL, such as data independence, query optimization and parallelization, can be inherited by AXL programs. Therefore, in the same way in which a database query can now be offloaded to a remote DB server, it will one day be possible to offload complex AXL programs to remote DB servers.

The parallelization of AXL programs provides another interesting direction for further work. While parallelization of aggregates along their group-by partition is easy to achieve, parallelization within the aggregate itself can be difficult achieve for complex aggregate functions [13]. The SQL statements used in AXL provide more opportunities for compiler analysis and automatic parallelization than procedural language statements, and this will provide a direction for future research.

Another interesting issue to be investigated is the relationship of AXL with procedural extensions for SQL, which are now being considered for standards with the aim of adding the power of procedural languages while retaining some of the benefits of SQL. These procedural extensions can be added to AXL in the future to overcome performance or expressiveness limitations encountered in actual applications. So far however, AXL proved sufficiently flexible and efficient for most database applications. In fact, AXL might also play a role in some data-intensive applications by replacing languages such as PL/SQL and JDBC used today.

We are currently working on several fronts. We are developing a test suite of data mining functions and database extenders written in AXL to validate the functionality and performance of our system. At the same time, we are completing and improving the AXL compiler, e.g., by adding more complete support for SQL data types, O-R extensions such as path notation, and a richer set of supporting indexes and storage structures. Issues such as better optimization and parallelization techniques for AXL aggregates provide other important topics for future research.

Acknowledgements

The authors are grateful to Hamid Pirahesh for suggesting a substantial editorial revision, and to the other referees for several improvements.

References

[1] R. Agrawal, R. Srikant. "Fast Algorithm for Mining Association Rules". In *VLDB'94*.

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

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

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

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

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

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

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

[9] Jeijun Kong, Cindy Chen and Carlo Zaniolo: A Temporal Extension of SQL for Object Relational Databases, submitted for publication, 2000.

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

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

[12] Informix: Datablade Developers Kid InfoShelf, Informix 1998, http://www.informix.co.za/answers/english/docs/dbdk/infoshelf/index.html

[13] G. S. Manku, S. Rajagopalan, B. G. Lindsay: Approximate Medians and other Quantiles in One Pass and with Limited Memory. SIGMOD Conference 1998: 426-435

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

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

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

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

[18] M. Stonebraker, L. Rowe, and M. Hirohama, "The Implementation of POSTGRES." *IEEE Transactions on Knowledge and Data Engineering*, **2**(1), March 1990.

[19] H. Wang, C. Zaniolo, "User-Defined Aggregates for Datamining," 1999 ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, Philadelphia, PA, May 30, 1999.

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

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

[22] Haixun Wang, Carlo Zaniolo, "User Defined Aggregates in Object-Relational Systems", in the 16th International Conference on Data Engineering (ICDE'2000), San Diego, USA, 2000.

[23] H. Wang and C. Zaniolo, User Defined Aggregates in Database Languages, 7th Int. Workshop on DB Programming Languages, Kinloch Rannoch, Scotland, Sept. 1999.

[24] C. Zaniolo, S. Ceri, C. Faloutzos, R. Snodgrass, V.S. S ubrahmanian, and R. Zicari, "Advanced Database Systems," Morgan Kaufmann Publishers, 1997.

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

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