User Defined Aggregates in Object-Relational Systems

Haixun Wang
Computer Science Department
University of California at Los Angeles
Los Angeles, CA 90095
hxwang@cs.ucla.edu

Carlo Zaniolo
Computer Science Department
University of California at Los Angeles
Los Angeles, CA 90095
zaniolo@cs.ucla.edu

Abstract

User-defined aggregates are essential in many advanced database applications, particularly in expressing data mining functions, but they find little support in current systems including Object-Relational databases. Three serious limitations of current systems are (i) the inability of introducing new aggregates (e.g., by coding them in procedural language as originally proposed in SQL3), (ii) the inability of returning partial results during the computation (e.g., to support online aggregation), and (iii) the inability of using aggregates in recursive queries (e.g., to express Bill of Materials and optimized graph searches). In this paper, we presents a unified solution to these problems which realizes SQL3 original proposal for user-defined aggregates (UDAs), and adds significant improvements in terms of expressive power and ease of use: in fact our SQL-AG system also supports online aggregation, monotonic aggregation, and a high-level aggregate definition language named SADL. We focus on applications of UDAs and SADL.

1. Introduction

The need for user-defined aggregates is pervasive in database applications and is particularly obvious in data mining, which we will therefore use frequently in our discussion. In fact, the demands posed by decision support applications have already motivated the introduction into commercial database systems of new aggregates such as rollups and data cubes. Yet, both researchers and vendors have overlooked User-Defined Aggregates (UDAs), which can play an even more critical and pervasive role in several application areas, including database-centric data mining applications. In this paper we show that

- Many data mining algorithms rely on specialized aggregates.
- The number and diversity of these aggregates implies that (rather than vendors adding ad hoc built-ins, which are never enough) a general mechanism should be provided to introduce new UDAs, in analogy to user-defined scalar functions that are now supported and widely utilized in Object-Relational DBMSs.
- UDAs can be easily and efficiently incorporated in Object-Relational DBMSs, to support UDA specs originally proposed for SQL3 [20], along with extensions for greater flexibility, functionality and usability.

2. Aggregates in datamining

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

Outlook	Temp	Humidity	Wind	PlayTennis
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. Tennis

The algorithm known as Boosted Bayesian Classifier [7] has proven to be the most effective at this task (in fact, it was the winner of the KDD'97 data mining competition). A

Naive Bayesian [7] 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 = 'PlayTennis'). 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:

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

The numerators and the denominators above can be easily computed using SQL aggregate queries. For instance, all the numerators values for the 4th column (the Wind column) can be computed as follows:

Example 1 *Using SQL's count Aggregate*

```
SELECT Wind, PlayTennis, count(*)
FROM Tennis
GROUP BY Wind, PlayTennis
```

Furthermore, the Super Groups construct now supported in various DBMSs [4], allows us to compute numerators and denominators using a single SQL query:

Example 2 *Using DB2's grouping sets*

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

In conclusion, this award-winning classification algorithm can be implemented well using the SQL `count` aggregate, thanks to the multiple grouping extensions recently introduced to support OLAPs. A database-centric approach to datamining is often preferable to main-memory oriented implementations, such as those used in [15], because it ensures better scalability and performance on large training sets. (The boosting step described in [15] can be implemented by simply increasing the weight of the misclassified tuples.) Unfortunately, unlike the Bayesian classifier just discussed, other datamining functions are prohibitively complex and inefficient to express and execute using the (SQL-compliant) data manipulation primitives of current database systems, and various research approaches were proposed to solve this problem [11, 5]. In this paper, we claim that enhancing the aggregate capabilities of database systems supplies the simplest and most cost-effective solution. Although the Bayesian classifier can be built using the aggregate functions found in current databases, more general data-mining algorithms require powerful User-Defined Aggregates (UDAs). Toward this goal, we implemented the UDA specifications originally proposed for SQL3 [20], which are not supported yet in current commercial systems, and extend them with the mechanism of early returns discussed in the next section.

3. UDAs and early returns

The recent OLAP extensions introduced in SQL were sufficient to express Bayesian classifier. But many advanced applications, including most data mining functions, cannot be supported well in SQL, even using ROLAP extensions. For instance the SPRINT classifier [14] chooses on which attribute and attribute value to split using a gini index:

$$gini(S) = 1 - \sum_{j=1}^{c} p_j^2 \tag{1}$$

Here p_j denotes the relative frequency of class j in S, which requires some counting. However, SPRINT handles continuous domains by first sorting the values and then evaluating all the counts incrementally for each new value in the sorted set. However, incremental evaluation of aggregates is not fully supported in current DBMSs (even when these provide support for rollups). Moreover, the purpose of the $gini$ index computation is to find the point where this index is minimum. Thus, for each new input value (i) the running count must be updated for each class, and (ii) the value of the gini function at this point must be calculated and compared with the current minimum to check if a new low is reached, and if so, the current value must be stored, since (iii) once all data have been visited, the minimum point for the gini must be returned, since this point might be used for the next split. Thus, the gini computations involves the following aggregate-like operations: (i) computing a running count, (ii) composing two aggregates (via the intermediate gini function), and (iii) returning the *point* where the minimum is found (rather than the *value* of that minimum). Unfortunately, none of these three operations can be easily expressed and efficiently supported in SQL2. However, the best splitting point using the gini index can be computed in a single pass through the dataset with the help of the UDAs originally proposed for SQL3 [20].

On the other hand, many important aggregate-like computations cannot be express using the UDAs proposed in [20]. For instance, time series analysis requires the computation of running sums or averages, and also of moving-window aggregates (e.g., total sales for the last four weeks) [10]. The incremental computation of these aggregates is very simple given that the data is stored as a temporal sequence. For instance, the running total of sales is computed by updating the sum at each new sale and returning the accumulated result at the end of each day. These problems can be solved by allowing aggregates to produce "early returns", i.e., to return values during the computation, rather than only at the end of the computation as in traditional aggregates. The computation of running sums, rollups, moving window averages, and many others aggregates becomes simpler and more efficient using the mechanism of *early returns*, which supports the generation of partial results while the computation of the aggregate is still in progress. For instance, while the absolute minimum of the $gini$ index must be produced as a 'final return', local minima and maxima should be generated as 'early returns' as soon as they are detected.

Early returns are also needed to support on-line aggregation [8]. On-line aggregation is very useful in many situations, e.g., to stop as soon as the computation of an average converges within the desired accuracy, or when aggregate such as count or sum have crossed the minimum support level (e.g., in the A Priori algorithm [1]).

On-line aggregates find many applications in datamining. For instance, in [15] a set of relations is given and the data miner searches for relations that satisfy some meta-level patterns specified by the user. For instance, the transitive composition pattern is:

$$P(X,Y) \wedge Q(Y,Z) \Rightarrow R(X,Z).$$

where P, Q and R stand for arbitrary database tables.

Then the system finds all the triplets in relations P, Q and R for which C_f exceeds a given confidence level C_f. (For instance, our three relations could be citizen, official_language and speaks.)

Now the level of confidence C_f in such a rule can be expressed as follows in SQL:

Example 3 *Level of confidence* C_f

SELECT **avg**(CASE WHEN **R.X=P.X** AND **R.Z=Q.Z**
THEN **1** ELSE **0** END)
FROM **P, Q, R**
WHERE **P.Y = Q.Y**

The on-line version of avg can be used to expedite this computation.

4. UDAs in SQL3 and SQL-AG

At UCLA, we have developed the SQL-AG [19] system that supports and extend the UDA specifications originally proposed for SQL3 [20]. 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 subquery is required to express this computation in standard SQL. But, following the approach proposed for SQL3 [9], which is similar to that used in Postgres [16], we can define the **minpoint** aggregate of Example 4. 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 of pair, for each successive pair, and when the end of stream is detected and the final values of the aggregate must be returned. 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**.

Example 4 *Defining an UDA to find the point of least value*

AGGREGATE FUNCTION **minpoint**(INTEGER, INTEGER)
RETURNS INTEGER
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;
}
```

The SQL-AG system developed at UCLA [19] extends the UDAs originally proposed for SQL3 by allowing the user to add to the aggregate specification a PRODUCE procedure to generate *early returns*, whereas the TERMINATE procedure produces *final returns*. For instance, to compute a traditional average, **myavg_terminate** will simply return the final sum divided by the final count, and **myavg_produce** will be replaced by NOP denoting that no early return is produced.

For an online aggregation, the average values will be returned every N records, where N is specified by a user or computed by a function that evaluates the rate of convergence. In this case, the aggregate might also produce a final return at the end, or **myavg_terminate** might just be specified as NOP. Early return extend the power and generality of UDAs a great deal. For instance, it allows us to express online aggregation [8] (and also as we shall see later monotonic aggregation). For instance, the following template defines an online version of average (e.g., one where a partial estimate is returned every 100 samples).

Example 5 *A UDA with Early Returns*

```
AGGREGATE FUNCTION myavg(value INTEGER)
RETURNS NUMBER (10.2)
STATE state_type
INITIALIZE myavg_single
ITERATE myavg_multi
PRODUCE myavg_produce
TERMINATE myavg_terminate
```

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 can depend on such order, inasmuch as users can write noncommutative functions in C. 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 overall algorithm is unaffected by this uncertainty in the result of the aggregate; for instance, classification algorithms work independent of which tie-breaking approach is taken for minima (don't care nondeterminism). In some applications, however, the presence or absence of a particular order for the data is important. 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 nonskewed data sets [8].

As we shall see later, SQL-AG provides the declarations that allow the user to specify the order, if any, in which data will be streamed through during the computation of the aggregate.

5. SADLE: a simple aggregate definition language

UDAs defined using a procedural languages such as C suffer from the same problems as the C-defined user defined functions [11]; these problems include the difficulty of developing and debugging applications, and a loss of of optimizability and parallelizability. An obvious solution to these problems consists in providing a high-level language for the definition of aggregates. In this paper, we present 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 directly in SQL. Although SQL limited expressive power can be source of problems, powerful UDAs can nevertheless be coded using SADL, as illustrated by the examples that follow.

Using SADL, the UDA **minpoint** of Example 4 can be defined as follows:

Example 6 *Defining* **minpoint** *in SADL*

```
AGGREGATE FUNCTION minpoint(point INTEGER,
                            value INTEGER)
RETURNS (rvalue INTEGER)
STATE (spoint INTEGER, svalue INTEGER)
INITIALIZE (VALUES(point, value))
ITERATE (SELECT CASE
            WHEN svalue > value THEN (point, value)
            ELSE (spoint, svalue) END CASE
         FROM state)
TERMINATE (SELECT spoint FROM state)
```

In Example 6, all parameters have explicit names (e.g., point and value) that can used in the SQL statements for INITIALIZE, ITERATE, PRODUCE and TERMINATE functions. These functions are expressed in SQL rather than in an external host language such as C. The STATE clause defines a relation, referred to as **state** in the aggregate routines, to hold the intermediate state information during the process of aggregation. For this example, the **state** relation contains the first (**point**, **value**) pair. The initial state of the aggregation is set by the INITIALIZE statement when the first record is processed. Then, ITERATE resets the state according to the new input and the previous state; for **minpoint**, it resets the state to the new input pair, if its **value** is less than that of the current state. Finally, the TERMINATE statement simply returns the spoint in the state relation. In many situations, there is also a PRODUCE statement to return results during the computation. The aggregate of Example 5 can now be re-defined using SADL; the WHERE clause in the PRODUCE statement ensures that online estimates for average are returned every 100 samples.

Name	*Salary*	*Title*	*DateOfBirth*	*Start*	*End*
Bob	60000	Asst. Provost	1945-04-09	1995-01-01	1995-06-01
Bob	70000	Asst. Provost	1945-04-09	1995-06-01	1995-10-01
Bob	70000	Provost	1945-04-09	1995-10-01	1996-02-01
Bob	70000	Professor	1945-04-09	1996-02-01	1997-01-01

Table 2. Employees with their salary history

Example 7 *Defining* myavg *in SADL*

```
AGGREGATE FUNCTION myavg(value INTEGER)
RETURNS (rvalue NUMBER (10.2))
STATE (sum INTEGER, count INTEGER)
INITIALIZE (VALUES(value, 1))
ITERATE (SELECT sum+value, count+1 FROM state)
PRODUCE (SELECT sum/count FROM state
                WHERE count MOD 100 =0)
TERMINATE (SELECT sum/count FROM state)
```

UDAs also find many applications in supporting temporal reasoning and queries. For instance, say that we want to retrieve the salary history of each employee from a temporal relation such as the one shown in Table 2 by projecting out the column *Title* and *DateOfBirth* from the relation. Then, for every employee, the maximal periods of time at each salary level must be computed by recursively merging overlapping intervals— an operation called *temporal coalescing* [23]. For example at hand, after projecting out *Title* and *DateOfBirth* we will have to coalesce the intervals of the last three tuples in Table 2, where Bob makes the same $70,000 salary. We obtain the following table:

Name	*Salary*	*Start*	*End*
Bob	60000	1995-01-01	1995-06-01
Bob	70000	1995-06-01	1997-01-01

Temporal coalescing is difficult to express in SQL [23]. For a time, it was thought impossible to express temporal coalescing completely in SQL; then an (inefficient) solution was discovered just a few years ago, involving complex, nested NOT EXISTS subclauses [23]. An efficient solution to the temporal projection problem is made possible by UDAs and SADL, as shown in Example 8. The SORTED BY clause in the example denotes that the data must be sorted attribute **istart**, the first parameter of the **coalesce** aggregate; this clause is discussed in more details in Section 6..

Since the intervals are sorted on the start date for each group, all we need to do is to detect whether there is a gap between the current interval and the interval we keep in the **state**. Whenever there is a gap, that is, the start date of the new interval is later than the end date of the current interval, the PRODUCE statement produces as early return the interval in **state**. Otherwise, the ITERATE statement coalesces the two intervals and store the result in **state**.

Example 8 *Temporal Coalescing* coalesce *in SADL*

```
AGGREGATE FUNCTION
    coalesce(istart DATE, iend DATE)
RETURNS (rStart DATE, rEnd DATE)
STATE (sStart DATE, sEnd DATE)
INITIALIZE (VALUES(istart, iend))
ITERATE (SELECT
   CASE WHEN sEnd ≥ istart THEN sStart ELSE istart
   END CASE,
   CASE WHEN iend ≥ sEnd THEN iend ELSE sEnd
   END CASE
          FROM state)
PRODUCE (SELECT sStart, sEnd FROM state
            WHERE sEnd < istart)
TERMINATE (SELECT sStart, sEnd FROM state)
SORTED BY istart
```

The UDA **coalesce** defined in Example 8 provides a general solution to the problem of supporting temporal projection in valid-time databases [23]. For example, to retrieve the salary history of each employee, we simply write the following query: (Note that the **coalesce** aggregate returns a complex data structure containing two columns: the start date and the end date.)

```
SELECT Name, Salary, coalesce(Start,End)
FROM employee
GROUP BY Name, Salary
```

Defining UDAs in SADL rather than in a procedural language provides substantial advantages. The simpler and more concise programs in which the user can now code UDAs represent only one of such benefits. In fact, a major simplification comes from the fact that the user need not worry about data type conversion and other “impedance mismtach” problems; SADL also inherits some of the benefits of SQL in terms of portability, interoperability, optimizability and parallelizability. Finally, UDAs defined in SADL will execute in the same address space as SQL without threatening its integrity (unlike unfenced UDFs written in C [4]).

6. SQL-AG

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 scratchpad UDFs to emulate the functionality of the corresponding U-DAs. 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(company)
FROM stock-closing
WHERE myavg_groupby(company,price)=1
```

Each UDA, named say **agg**, is implemented with two automatically generated DB2 UDFs, namely, $agg\_groupby$ and agg. The UDF $agg\_groupby$ performs the actual computation and should be applied on every record for aggregation. It uses a hash table to keep the aggregate value of each group. If the PRODUCE routine is also defined for agg, then $agg\_groupby$ will return 1 when $agg\_produce$ wants to produce an early return for this record. The UD-F $agg(group)$ retrieves from the hash table the last value computed by $agg\_groupby$ for $group$. A detailed description of SQL rewriting can be found in [19].

SQL-AG allows the user to specify that the implementation of an aggregate sort-based rather than hash-based through the SORTED BY GROUPBY option, using the following syntax:

SORTED BY [GROUPBY] $[arg_1, \cdots, arg_n]$

Therefore, the data can be sorted on (i) group-by attributes, (ii) on a list of other attributes or both. The knowledge that data is sorted in a certain way is essential for a user who defines the UDA; for instance the coalesce aggregate of Example 8 works under the assumption that the data is sorted by istart. Furthermore if the GROUPBY option is used, then the SQL-AG system performs a sort-based aggregate, where, upon detection of a new value for the groupby attributes, the aggregate for the old group is produced and the computation for the new one begins.

Performance We compared the performance of native D-B2 builtins against SQL-AG UDAs on a Ultra SPARC 2 with 128 megabytes memory. We used a new UDA, `myavg`,

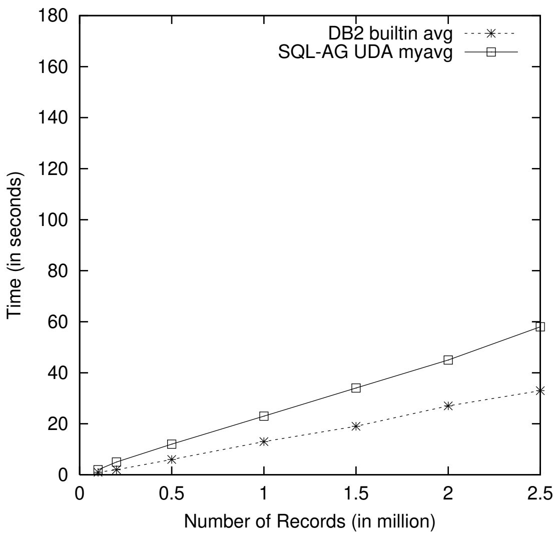

Figure 1. Aggregates without Group-by

which has the same functionality as the builtin aggregate `avg`. Figure 1 shows that when aggregation contains no group-by columns, there is a slight performance penalty resulted from calling UDFs. However, when group-by columns exist (Figure 2), UDAs of SQL-AG outperform D-B2's builtin aggregates[1]. This is due to the fact that DB2 implements grouping by pre-sorting all the records, while SQL-AG uses hashing. Similar results hold for the other builtins. Thus the flexibility of supporting early returns is not bought at the price of lesser performance.

In fact, there are several situations where UDAs can be used to enhance the performance of aggregate queries already supported by the current SQL. For instance, all counts in Example 2 can be computed in one pass through the data using a hash based join. However, in DB2 and other commercial systems, an implementation of GROUPING SETS normally results in a cascade of sorting operations. As illustrated by Figure 3, the differences can be substantial.

7. Monotonic aggregation

The introduction of early returns provides a surprisingly simple solution for the problem of monotonic aggregation that had remained open for many years [13]. Indeed, while SQL3 and the new generation of Object-Relational systems support recursive queries, they require all queries to be stratified with respect to negation and set aggregates due to the nonmonotonic nature of these constructs [20]. As results of these limitations, applications such as Bill of Materials (BoM) and graph traversal that require the use of

[1] These results were obtained using DB2 UDFs in an unfenced mode [4]. Execution in the fenced mode was considerably slower.

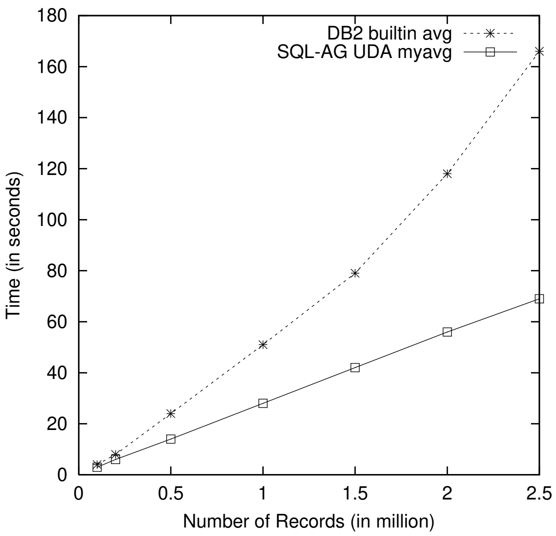

Figure 2. Aggregates with Group-by

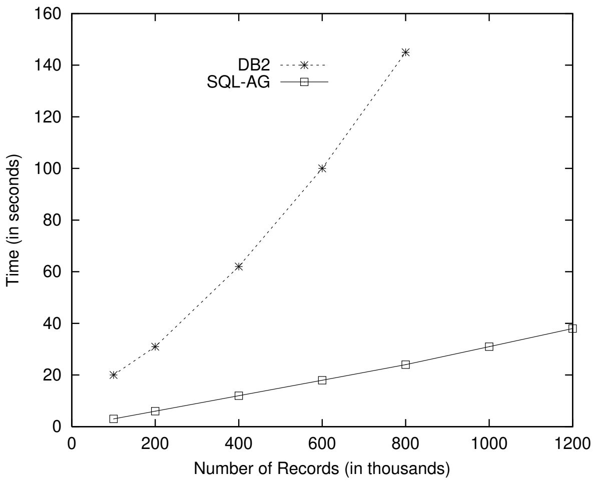

Figure 3. DB2's grouping set vs. SQL-AG's UDA

aggregates the recursive queries are still not supported well in SQL. Many research efforts have failed produce practical solutions to this problem. For instance the solution proposed in [13] shows that some aggregates are monotone in special lattices that are different from the standard set containment lattice. Unfortunately, we do not have techniques for automatically recognizing these lattices and adjusting the fixpoint computation accordingly [17].

However, as we have recently shown [21], aggregates with only early returns (i.e., those declared with 'TERMINATE NOP') are monotonic with respect to set-containment. This monotonicity test is trivial to implement. Furthermore, monotonic aggregates can be freely allowed in recursive queries, and no modification is required in current SQL systems or deductive databases except excusing monotonic aggregates from the stratification condition. Monotonic aggregates are fully supported in SQL-AG and in $\mathcal{LDL}$++ version 5.0 [22].

These monotonic aggregates have then been used [19] to express complex algorithms for BoM, graph traversal and greedy optimization, such as Dijkstra's single source shortest path, which, in the past were considered beyond the realm of SQL queries.

A monotonic aggregate of particular interest is **mcount**:

Example 9 *Monotonic aggregate* **mcount**

```
AGGREGATE FUNCTION mcount(value ANY)
RETURNS (count INTEGER)
STATE (sCount INTEGER)
INITIALIZE (VALUES(1))
ITERATE (SELECT sCount+1 FROM state)
PRODUCE (SELECT sCount FROM state)
TERMINATE NOP
```

For example, assume that we have the **mcount** aggregate, which through the mechanism of early returns produces a new partial count for each new element in the set. Thus, for a set with cardinality 4, **mcount** will simply produce $1, 2, 3, 4$; when a new element is added to the set **mcount** produces $1, 2, 3, 4, 5$, thus achieving *monotonicity with respect to set-containment*. (Observe, that **mcount** is monotonic and deterministic; a **msum** aggregate returning the sum so far is still monotonic, but nondeterministic.)

We now express several examples derived from [13] using our new monotonic aggregates.

Join the Party Some people will come to the party no matter what, and their names are stored in a **sure** relation. But others will join only after they know that at least $K = 3$ of their friends will be there. Here, a pair **(Pname, Fname)** in **friend** denotes that **Fname** is **Pname**'s friend. Suppose the **sure** and **friend** relation contain the following tuples:

sure
Pname
mark
tom
jane

friend	
Pname	**Fname**
jerry	mark
penny	mark
jerry	jane
penny	jane
jerry	penny
penny	tom

Figure 4. Join the Party

To find all the people that will eventually come to the party, we use the mcount aggregate in the following query:

Example 10 *Join the Party using Monotonic Aggregate* mcount

```
WITH RECURSIVE wllcm(Name) AS
(  SELECT Pname FROM sure
UNION ALL
   SELECT f.Pname
   FROM wllcm, friend f
   WHERE wllcm.Name = f.Fname
   GROUP BY f.Pname
   HAVING mcount(f.Fname)=3 )
SELECT Name
FROM wllcm
```

The above recursive query yields the following solution:

Name
mark
tom
jane
penny
jerry

Next, we define aggregate **mmin**, a monotone version of the standard aggregate **min**. UDA **mmin** returns the least value seen so far.

Example 11 *Monotonic aggregate* **mmin**

```
AGGREGATE FUNCTION mmin(value INTEGER)
RETURNS (value INTEGER)
STATE (Cmin INTEGER)
INITIALIZE (VALUES(value))
ITERATE (SELECT CASE WHEN value < Cmin
                THEN value ELSE Cmin
                END CASE
           FROM state)
PRODUCE (SELECT value FROM state
                WHERE value < Cmin)
TERMINATE NOP
```

Observe that we have also added conditions in the PRODUCE clause to limit the production of redundant returns.

We next consider a simple approach on how to compute least distance paths in a graph. More sophisticated approaches based on greedy algorithms are discussed in [19].

Least-Distance Connections For instance, say that the table mygraph, with attributes **(Start, End, C)** represents the arcs in a graph, where **C** is the cost the arc from node **Start** to node **End**, then the least-cost path between any two nodes can be computed as shown in Example 12.

Example 12 *Least-Distance Connections using the Monotonic Aggregate* **mmin**

```
WITH RECURSIVE ld(Start, End, C) AS
(  SELECT Start, End, mmin(C) FROM mygraph
   GROUP BY Start, End
UNION ALL
   SELECT ld1.Start, ld2.End, mmin(ld1.C+ld2.C)
   FROM ld ld1, ld ld2
   WHERE ld1.End=ld2.Start
   GROUP BY ld1.Start, ld2.End)
SELECT Start,End,min(C)
FROM ld
GROUP BY Start,End
```

This transitive-closure like computation adds a new arc **(Start, End, C)** to **ld** provided that this then becomes the new least-cost arc between the nodes **Start** and **End**. The arcs so produced are then used in the next step of the semi-naive computation. At the end of this fixpoint computation, we need to select the actual least-cost arcs between these two nodes, out of the succession of arcs of decreasing **C** values produced in the computation. Let us compare this query to the standard stratified query which generates all possible paths between pairs, from which least-cost paths are then selected in the last step. In the situation where there are cycles of positive cost, our mmin program terminates whereas the other does not. A lower cost is also expected when there is no cycle: for instance, in any graph where longer paths correspond to higher costs, then the cost of our computation reduces to that of producing all pairs of nodes (as in a transitive closure) while the stratified program will produce all paths between pairs of nodes.

For a given graph, the values obtained during the computation of **ld** can vary depending on the order in which the arcs are considered. The final values are, however, the same (a nondeterministic computation producing a deterministic answer).

Company Control Another interesting example is transitive ownership and control of corporations. Say that a tuple **(C1, C2, Per)** in the relation **owns** denotes corporation **C1** owns **Per** percent of the shares of corporation **C2**. Then, **C1** controls **C2** if it owns more than 50% of its shares. In general, to decide whether **C1** controls **C3** we must also add the shares owned by corporations such as **C2** that are controlled by **C1**. This yields the transitive control program of Example 13. Thus, every company controls itself, and a company **C1** that has transitive ownership of more than 50% of **C2**'s shares controls **C2**. In the last rule, **transitive_own** computes transitive ownership with the help of **msum** that adds up the shares of controlling companies. Observe that any pair **(c.C1, o.C2)** is added at most once to **control**, thus the contribution of **c.C2** to **c.C1**'s transitive ownership of **o.C2** is only accounted once.

Example 13 *Company Control Problem using Monotonic Aggregate* **msum**

```
WITH RECURSIVE
  control(C1, C2) AS
    ( SELECT C1, C1 FROM owns
    UNION ALL
      SELECT t.Owner, t.Company
      FROM transitive_own t
      WHERE TotalPercent > 50 ),
  transitive_own(Owner, Company, TotalPercent) AS
    ( SELECT c.C1, o.C2, msum(o.Per)
      FROM control c, owns o
      WHERE c.C2 = o.C1
      GROUP BY c.C1, o.C2)
SELECT C1, C2
FROM control
```

To further simplify the program and expedite the computation we can introduce a special (deterministic) aggregate **sum50** as follows:

Example 14 *Monotonic aggregate* **sum50**

```
AGGREGATE FUNCTION sum50(value INTEGER)
RETURNS (value INTEGER)
STATE (sum INTEGER)
INITIALIZE (VALUES(value))
ITERATE (SELECT sum+value FROM state
         WHERE sum < 50)
PRODUCE (SELECT 1 FROM state
          WHERE sum ≥ 50)
TERMINATE NOP
```

Then, the Company Control problem can be coded as follows:

Example 15 *Company Control Problem using Monotonic Aggregate* **sum50**

```
WITH RECURSIVE cntrl(C1, C2) AS
  ( SELECT C1, C1, 1 FROM owns
  UNION ALL
    SELECT o.C1, o.C2
    FROM owns o WHERE o.Per > 50
  UNION ALL
    SELECT c.C1, o.C2, sum50(o.Per)
    FROM cntrl c, owns o WHERE c.C2=o.C1
    GROUP BY c.C1, o.C2)
SELECT C1, C2
FROM cntrl
```

Thus, the **sum50** aggregate succeeds only when the 50% threshold is crossed during the summation. Here, the value of 50 was cast into the very definition of our aggregate. Alternatively, this value could be given as an additional parameter.

Bill-of-Materials (BoM) Applications BoM applications represent an important application area that requires aggregates in recursive rules. Say, for instance that a tuple **(Part, SubPart, Qty)** in **assembly** denotes that **Part** contains **SubPart** in quantity **Qty**. We also have elementary parts described by the relation **basic_part** with columns **Part**, **Price**. Then, the following program computes the cost of a part as the sum of the cost of the basic parts it contains.

Example 16 *Bill-of-Materials using Monotonic Aggregate* mcount *and* msum

```
WITH RECURSIVE cost(Part, nSubParts, Cost) AS
  ( SELECT Part, 0, Price
    FROM basic_part
  UNION ALL
    SELECT a.Part, mcount(c.Part),
           msum(c.Cost×a.Qty)
    FROM assembly a, cost c, prolfc p
    WHERE c.Part = a.SubPart
      AND c.Part = p.Part
      AND c.nSubParts = p.nSubParts
    GROUP BY a.Part)
SELECT Part, Cost FROM cost
```

Thus, the key condition in the body of the recursive rule is that a subpart **Part** is counted in **cost** only when all **Part**'s children have been counted. This occurs when the number of **Part**'s children counted so far by **mcount** is equal to the out-degree of this node in the graph representing `assembly`. This number is kept in a prolificacy table **prolfc** (with columns **Part, nSubParts**) that can be computed as follows:

```
CREATE VIEW prolfc(Part, nSubParts) AS
  SELECT Part, count(SubPart) FROM assembly
  UNION
  SELECT Part, 0 FROM basic_part
```

The technique of counting the children could also be used with least-distance problem of Example 12, if the underlying graph is acyclic. For cyclic graphs, we must use the current formulation that exploits the property that extrema are unaffected by duplicates (idempotency).

8. Conclusions

User-defined aggregates represent a powerful addition to SQL that will greatly extends its ability of dealing with advanced application domains, particularly decision support and data mining. In this paper, we have described the SQL-AG system that supports UDAs on top of DB2; the good performance of SQL-AG shows that UDAs dovetail perfectly with current Object-Relational systems and can be added to these systems with relatively minor effort. Furthermore, in SQL-AG, we have extended the UDA constructs originally proposed for SQL3 to provide:

- support for early returns,
- support for monotonic aggregation,
- a simple high-level language, SADL, that simplifies the definition of UDAs.

We have also discussed the critical role that extended UDAs can have in advanced application domains, such as temporal databases [3] and data mining [18]. We are currently exploring SADL extensions for more complex datamining functions [1, 12, 14], and for applications that currently use commercial DataBlades packages.

References

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

[2] S. Brin, R. Motwani, J.D. Ullman, S.L. Tsur, "Dynamic Itemset Counting and Implication Rules for Market Basket Data". In *SIGMOD'97*.

[3] C. Chen, C. Zaniolo, "Universal Temporal Extensions for Data Languages". in The Fifteenth International Conference on Data Engineering, Sydney, Australia, March, 1999.

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

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

[6] D. Chatziantoniou and K. A. Ross. "Querying Multiple Features of Groups in Relational Databases". *VLDB 1996*.

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

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

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

[12] R. Rastogi, K. Shim. "PUBLIC: A Decision Tree Classifier that Integrates Building and Pruning". VLDB 1998: 404-415

[13] K. A. Ross and Yehoshua Sagiv, "Monotonic Aggregation in Deductive Database", JCSS 54(1), 79-97 (1997) .

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

[15] W. Shen, K.Ong, B. Mitbander, C. Zaniolo, "Metaqueries for Data Mining", in *Advances in Knowledge Discovery and Data Mining*, U. M. Fayyad et al (eds.), MIT Press, 1996.

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

[17] A. Van Gelder. Foundations of Aggregations in Deductive Databases In *Proc. of the Int. Conf. On Deductive and Object-Oriented d atabases*, 1993.

[18] 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.

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

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

[21] C. Zaniolo, H. Wang, "Logic-Based User-Defined Aggregates for the Next Generation of Database Systems". in The Logic Programming Paradigm: Current Trends and Future Directions, K.R. Apt, V. Marek, M. Truszczynski, D.S.Warren (eds.), Springer Verlag, 1998.

[22] C. Zaniolo et al., $\mathcal{LDL}$++ 5.0 "http://www.cs.ucla.edu/ldl/"

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