User-Defined Aggregates in Database Languages

Haixun Wang and Carlo Zaniolo

Computer Science Department
University of California at Los Angeles
{hxwang,zaniolo}@cs.ucla.edu

Abstract. User-defined aggregates (UDAs) can be the linchpin of sophisticated data mining functions and other advanced database applications, but they find little support in current database systems. In this paper, we describe the SQL-AG prototype that overcomes these limitations by supporting UDAs as originally proposed in Postgres and SQL3. Then we extend the power and flexibility of UDAs by adding (i) early returns, (to express online aggregation) and (ii) syntactically recognizable monotonic UDAs that can be used in recursive queries to support applications, such as Bill of Materials (BoM) and greedy algorithms for graph optimization, that cannot be expressed under stratified aggregation. This paper proposes a unified solution to both the theoretical and practical problems of UDAs, and demonstrates the power of UDAs in dealing with advanced database applications.

1 Introduction

The importance of new specialized aggregates in advanced applications is exemplified by rollups and data cubes that, owing to their use in decision support applications, have been included in all new releases of commercial DBMSs. Yet, we claim that database vendors, and to a certain extent even researchers, have overlooked User-Defined Aggregates (UDAs), which can play an even more critical and pervasive role in advanced database applications, particularly data mining. In this paper, we show that:

- ⋄ Many data mining algorithms rely on specialized aggregates.
- ⋄ The number and diversity of these aggregates imply 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 of object-relational (O-R) DBMSs.
- ⋄ UDAs can be easily and efficiently incorporated in O-R DBMSs, in accordance with the UDA specs originally proposed in SQL3 [8]. This is also true for the UDA extensions discussed in this paper that greatly improve their flexibility and functionality.

2 Aggregates in Data Mining

As a first example, 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.

Table 1. Tennis

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

The algorithm known as Boosted Bayesian Classifier [5] 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* [5] 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 maximizes 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 third 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 contained in the recent OLAP extensions of commercial SQL systems[3] allows us to express this computation in a single query:

Example 2. Using DB2's grouping sets

```
SELECT Outlook, Temp, Humidity, Wind,
                        PlayTennis, count(*)
FROM Tennis
GROUP BY GROUPING SETS (PlayTennis),
           ((Outlook, PlayTennis), (Temp, PlayTennis),
           (Humidity, PlayTennis), (Wind,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 data mining is often preferable to main-memory oriented implementations, because it ensures better scalability and performance on large training sets. Unfortunately, unlike the Bayesian classifier just discussed, most data mining functions are prohibitively complex and inefficient to express and execute using the (SQL-compliant) data manipulation primitives of current database systems [23]. In this paper, we claim that the simplest and most cost-effective solution to this problem consists in adding powerful UDA capabilities to DBMSs. Toward this goal, we implemented the UDA specifications originally proposed for SQL3 [8], (but not supported yet in commercial systems) and extended them with the mechanism of early returns discussed in the next section. While we use mostly data mining examples, UDAs are needed in many applications to overcome the limited expressive power of SQL; for instance, we found them essential in implementing temporal database queries [4].

3 UDAs and Early Returns

While the aggregate computations needed in a Bayesian classifier can be expressed using SQL built-ins, this is not the case for most data mining algorithms. For instance the SPRINT classifier [24] chooses on which attribute and value to split next 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 the training set S. For discrete domains (i.e., categorical attributes) this operation can be implemented using the standard count aggregate of SQL. However, the attribute values from continuous domains must be first sorted on the attribute value, and then the count must be evaluated incrementally for each new value in the sorted set. Now, incremental evaluation of aggregates is not fully supported in current DBMSs (even those providing support for rollups). Moreover, the objective of the *gini* computation is to select a point (and a column from the table) where the gini index is minimum. Thus, for each new value in the sorted set, (i) the running count for each class must be updated, and (ii) the value of the gini function at this point must be calculated and compared with the minimum so far, to see if the old value must be replaced with the new one; in fact, after every value has been examined, (iii) the minimum point for the gini must be returned, since this point will be used for the next split. Therefore, 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). None of these three operations can be easily expressed and efficiently supported in SQL2; but with UDAs originally proposed for SQL3 [8], they can be merged into a single and efficient computation that determines the splitting point in a single pass through the dataset.

While UDAs such as those proposed for SQL3 [8] are the right tool for computing a gini index, they cannot express many other aggregate computations, and, in particular, they cannot express online aggregation [6]. 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 aggregates, such as count or sum, have crossed the minimum support level (e.g., in the A Priori algorithm). On-line aggregates find many applications in data mining [26], and greatly extend the power of UDAs.

We can solve these problems by allowing UDAs 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 rollups, running aggregates, moving window aggregates, and many others becomes simple and efficient using the mechanism of *early returns*, which allows the generation of partial results while the computation of the aggregate is still in progress [4].

For instance, while final returns can be used to find a point of global minimum for a function, such as the gini function, early returns will be used to compute the points where local extrema occur (i.e., the valleys and the peaks).

4 Extended UDA and SQL3

In this section, we discuss the SQL-AG language, whereas the the SQL-AG system is described in the next section. To introduce a UDA named myavg, according to the specifications proposed for SQL3 [8], we must proceed as shown in Example 3. Basically, the user must define three user-defined functions (UDFs) for the three cases INITIALIZE, ITERATE, and TERMINATE. The INITIALIZE (ITERATE) function defines how the first (successive) values in the set are processed. The TERMINATE function describes the final computation for the aggregate value. Thus, to compute the traditional average, the state will hold the variables sum and count; these are, respectively, initialized to the the first value of the set and to 1 by myavg_single. Then, for each successive value in the set, myavg_multi adds this value to sum and also increases count by 1. Finally, myavg_terminate returns sum/count.

Example 3. A UDA Definition

```
AGGREGATE FUNCTION myavg( IN NUMBER)
RETURNS NUMBER
STATE state
INITIALIZE myavg_single
ITERATE myavg_multi
TERMINATE myavg_terminate
```

The search for global minima for the gini index can be easily programmed using two UDFs gini-single and gini-multi. But, in the presence of ties, the gini-terminate function will return any of the points where the global minimum occurs, e.g., the first point. Therefore, the order in which the elements of a set are considered becomes important, and can influence the final result, and to the extent that this order is unknown, UDAs display a *nondeterministic* behavior. Traditional SQL built-ins are instead deterministic, i.e., they always return the same result on a given set. This nondeterministic behavior is not an impediment in formalizing the logic-based semantics of UDAs, and in writing effective queries; in fact, nondetermism is a critical feature in many real life applications.

An important extension introduced by SQL-AG is *early returns* that are specified using a PRODUCE myavg_produce function. For instance, with an online aggregation, the average of values computed so far can be returned every N records, where N is specified by a user or computed by a function that evaluates the rate of convergence. Early returns are useful in many other roles, besides online aggregation. For instance, in a time series we need to find local extrema, i.e., valleys and peaks, which are easily handled with early returns. In this case, the aggregate might not produce any final return, and this can be specified by TERMINATE NOP.

An important issue brought to a resolution by early returns is that of monotonicity: in the next section we prove that aggregates with only early returns (i.e., those declared with TERMINATE NOP) are monotonic and can be freely used in recursion. This provides a surprisingly simple solution to the problem of detecting monotone aggregation [15] that had remained open since Ross and Sagiv demonstrated the many useful applications of these aggregates [19]. Monotonic aggregates can be used to express graph traversal algorithms, greedy algorithms, Bill of Materials (BoM) applications and other computations that were previously viewed to be beyond the capabilities of SQL and Datalog [19,9,17,10].

For example, say that we have defined a mcount aggregate, where PRODUCE returns a new partial count for each new element in the set, and thus there is no final return. Therefore, mcount is a monotonic aggregate: for a set with cardinality 4, mcount will simply produce $1, 2, 3, 4$; when a new element is added to the set mcount returns $1, 2, 3, 4, 5$. Thus mcount is monotonic with respect to set containment, whereas the traditional count returns first $\{4\}$ and then $\{5\}$, where the latter set is not a superset of the former. (Observe, that mcount is monotonic and deterministic; the msum aggregate returning the sum so far is still monotonic, but nonderministic.)

Consider now the use of monotonic aggregates in solving recursive problems. The **Join-the-Party** problem states that some people will come to the party no matter what, and their names are stored in a sure(Person) relation. But others will join if at least three of their friends will be there. Here, friend(P, F) denotes that P regards F as a friend. A monotonic user-defined aggregate mcount is used inside a recursive query to solve this problem. The PRODUCE routine of mcount returns the intermediary count and its TERMINATE routine is defined as NOP.

Example 4. Join the Party in SQL-AG

```
WITH RECURSIVE willcome(Name) AS
(   SELECT Person FROM sure
UNION ALL
    SELECT f.P
    FROM willcome, friend f
    WHERE willcome.Name = f.F
    GROUP BY f.P
    HAVING mcount(f.F)=3
) SELECT Name FROM willcome
```

As we shall see later, this program has a formal logic-based semantics, inasmuch as it can be translated into an equivalent Datalog program that has stable model semantics [11]. On a more practical note, a host of advanced database applications, particularly data mining applications, benefit from our UDAs. For instance, it is possible to express complex algorithms such as the ‘A Priori’ algorithm using the monotonic version of count, resulting in more flexibility and opportunities for optimization. Since the result of a fixpoint computation on monotonic operators is not dependent on the particular order of execution, several variations of A Priori are possible; for instance, a technique where the computation of item-sets of cardinality $n+1$ starts before that of cardinality n is completed was proposed in [2]. We were also able to implement other data mining algorithms, such as SPRINT/PUBLIC(1) and iceberg queries [7] in SQL-AG, with very little effort. The UDAs were used here to build histograms, calculate the gini index, and to perform in one pass the complex comparisons of tree costs needed to implement PUBLIC(1) [18].

5 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 UDAs [3]. For instance, say that we want to find the average salary of employees by department, using the UDA myavg, instead of the SQL built-in; then we can write:

```
SELECT dept, myavg(salary)
FROM emp
GROUP BY dept
```

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

```
SELECT dept, myavg(dept)
FROM emp
WHERE myavg_groupby(dept, salary)=0
GROUP BY dept
```

Here, the funtion myavg_groupby performs the actual computation of the aggregate by applying to each record the INITIALIZE and ITERATE functions written by the user (i.e., the functions **myavg_single** and **myavg_multi** for Example 3), and then returning 0. Finally, for each dept the function myavg(dept) applies the TERMINATE function written by the user (i.e., **myavg_terminate** for Example 3) to the last values computed by myavg_groupby, returning the final result. Similar transformations are used to handle the case where the UDA only has early returns, and the more complex case where both early returns and final returns are used. More details about SQL-AG and its implementation can be found in [25].

We compared the performance of native DB2 builtins against SQL-AG UDAs on a Ultra SPARC 2 with 128 megabytes memory. We used a new UDA, `myavg`, which has the same functionality as the builtin aggregate `avg`. Figure 1 shows that, when aggregation contains no group-by columns, our UDAs incur in a modest performance penalty with respect to DB2 builtins. However, when group-by columns are used, then the UDAs of SQL-AG normally outperform DB2's builtin aggregates, as shown in Figure 2. This is due to the fact that DB2 implements grouping by pre-sorting all the records, while SQL-AG uses hashing. This advantage is lost if the group-by columns coincide with the primary key for the relation at hand, and thus the data is already in the proper order. In this case, our UDAs are somewhat slower than DB2 builtins—bottom curve in Figure 2.

Our performance comparison shows that, in general, user-defined aggregates can be expected to have performance comparable to that of builtins [1]. In fact,

[1] These results were obtained using DB2 UDFs in an unfenced mode [3]. Execution in the fenced mode was considerably slower.

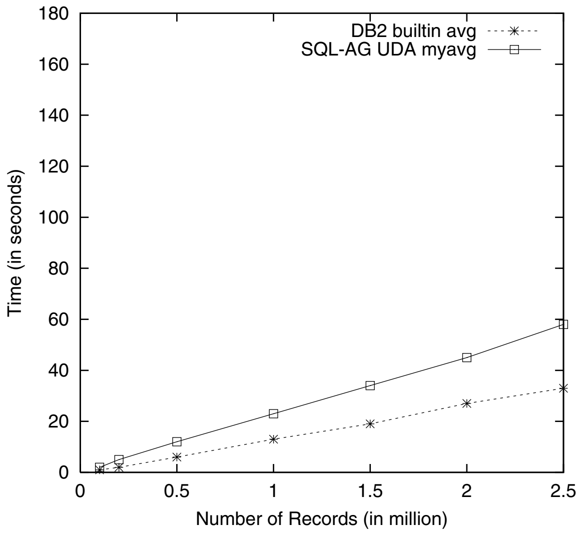

Fig. 1. Aggregates without Group-by

there are several situations where specialized UDAs will be preferred to builtin aggregates simply for performance reasons. For instance, all counts needed in Example 2 can be computed in one pass through the data using a hash-based approach (and SQL-AG allows the user to specify whether the implementation of each aggregate is hash-based or sort-based). In DB2, and other commercial systems, an implementation of GROUPING SETS normally results in a cascade of sorting operations. As illustrated by Figure 3, this resulted in a substantial speed-up, and improved scalability (DB2 on our workstation refused to handle more than 800000 records).

6 Aggregates in Logic

The procedural attachments used to define new aggregates in SQL-AG could leave the reader with the impression that these are merely procedural extensions, without the benefits of the formal logic-based semantics that provides the bedrock for relational query languages and the recent SQL extensions for recursive queries. Fortunately, this is not the case, and we next provide a logic based formalization for UDAs. This also yields a simple syntactic characterization of aggregates that are monotonic in the standard lattice of set-containment, and can therefore *be used without restrictions in recursive queries*. This breakthrough offers a simple solution to the monotonic aggregation problem, and allows us to express applications such as BoM and graph traversals that had long been problematic for SQL and Datalog [19,10,15].

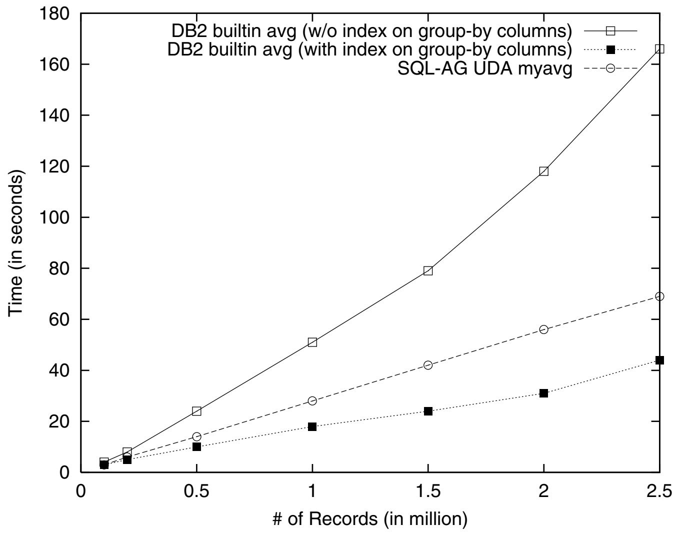

Fig. 2. Aggregates with Group-by Columns

Inductive Definition of Aggregates. Aggregate functions on (non-empty) sets can be defined by induction. The base case for induction is that of singleton sets; thus, for count, sum and max, we have $count(\{y\}) = 1$, $sum(\{y\}) = y$, and $max(\{y\}) = y$. Then, by induction, we consider sets with two or more elements; these sets have the following form: $S \sqcup \{y\}$, where $\sqcup$ denotes disjoint union (thus S is the "old" set while y is the "new" element). Then, our *specific inductive functions* are as follows: $sum(S \sqcup \{x\}) = sum(S) + x$, $count(S \sqcup \{x\}) = count(S) + 1$, $max(S \sqcup \{x\}) =$ `if` $x > max(S)$ `then` x `else` $max(S)$. Thus, expressing aggregates in Datalog can be broken down in two parts: (i) writing the rules for the specific inductive functions used for this particular aggregate, and (ii) writing the recursive rules that enumerate the elements of a set one-by-one as needed to apply the specific inductive functions. Part (i) is described next, and part (ii) is discussed in the next section. For concreteness, we use here the syntax of $\mathcal{LDL}$++ [20,28].

In $\mathcal{LDL}$++, the base base step in the computation of an aggregate is expressed by `single` rules that apply to singleton sets, while the induction step is expressed by `multi` rules that apply to sets with two or more elements. Thus, we obtain the following definitions for `sum`

```
single(sum, Y, Y).
multi(sum, Y, Old, New) ← New = Old + Y.
```

and for `max`

```
single(max, Y, Y).
multi(max, Y, Old, Y) ←     Y > Old.
multi(max, Y, Old, Old) ← Y <= Old.
```

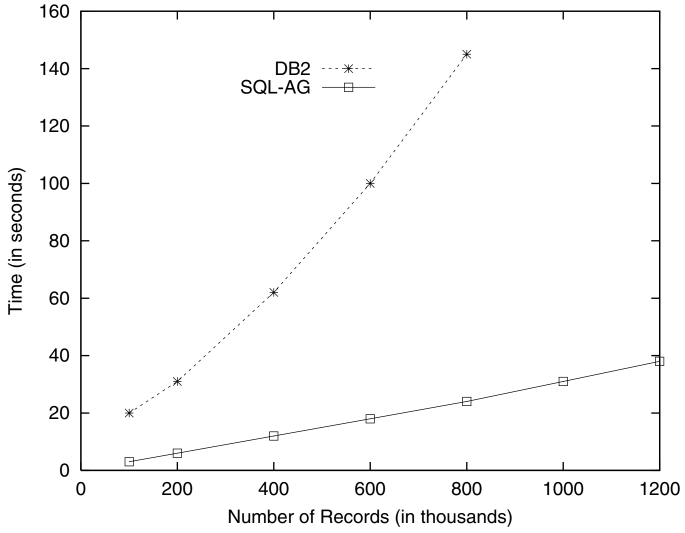

Fig. 3. DB2's grouping set vs. SQL-AG's UDA

Therefore, we use the first argument in the heads of the rules to hold the unique name of each aggregate.

Then, the $freturn$ rules are used to specify the value to be returned at the end of the computation. For `sum` and `max` the return rules are as follows:

$$\texttt{freturn}(\texttt{sum}, \texttt{Y}, \texttt{Old}, \texttt{Old}). \quad \texttt{freturn}(\texttt{max}, \texttt{Y}, \texttt{Old}, \texttt{Old}).$$

The complete definition of the aggregate `avg` is as follows:

$$\begin{array}{ll}
\texttt{single}(\texttt{avg}, \texttt{Y}, (\texttt{Y}, \texttt{1})). & \\
\texttt{multi}(\texttt{avg}, \texttt{Y}, (\texttt{Sum}, \texttt{Count}), (\texttt{Nsum}, \texttt{NCount})) \leftarrow & \texttt{Nsum} = \texttt{Sum} + \texttt{Y}, \\
 & \texttt{Ncount} = \texttt{Count} + \texttt{1}. \\
\texttt{freturn}(\texttt{avg}, \texttt{Y}, (\texttt{Sum}, \texttt{Count}), \texttt{Avg}) \leftarrow & \texttt{Avg} = \texttt{Sum}/\texttt{Count}.
\end{array}$$

The $\mathcal{LDL}$++ extension recently developed at UCLA also supports *early returns*, which must be specified using `ereturn` rules. Thus, if the user wants to see partial results from the computation of averages every 100 elements, the following rule must be added:

$$\begin{array}{l}
\texttt{ereturn}(\texttt{avg}, \texttt{X}, (\texttt{Sum}, \texttt{Count}), \texttt{Avg}) \leftarrow \\
\qquad\qquad \texttt{Count mod 100} = \texttt{0}, \texttt{Avg} = \texttt{Sum}/\texttt{Count}.
\end{array}$$

In order to find the average salary of employees grouped by department, the user can thus write:

$$\texttt{p}(\texttt{DeptNo}, \texttt{avg}\langle\texttt{Sal}\rangle) \leftarrow \texttt{empl}(\texttt{Ename}, \texttt{Sal}, \texttt{DeptNo}).$$

Thus, in this syntax, that is shared by both $\mathcal{LDL}$++ [28] and CORAL [17], aggregates, such as $\texttt{avg}\langle\ldots\rangle$, are used as arguments in the head of the rule, and the remaining non-aggregate arguments are interpreted as group-by attributes.

The head aggregate construct can be viewed as a meta-level construct with first order semantics; as shown in the next section, it can be expanded into the internal rules that, along with the single, multi, freturn and ereturn rules written by the user, express the formal meaning of aggregates in logic.

Let us now define a logic-based equivalent of the SQL-AG program of Example 4. We begin by defining `mcount` that returns the incremental count at each step:

```
single(mcount, Y, 1).
multi(mcount, Y, Old, New) ←     New = Old + 1.
ereturn(mcount, Y, Old, New) ← Old = nil, New = 1.
ereturn(mcount, Y, Old, New) ← Old ≠ nil, New = Old + 1.
```

The first `ereturn` rule applies when `Old` = `nil` (where `nil` is just a special value—not the empty list). Now, the condition `Old` = `nil` is only satisfied when the first `Y` value in the set is found; thus, this rule is enabled together with `single` rule, and produces the integer 1. After that, the second `ereturn` rule applies repeatedly, in parallel with the `multi` rule, producing $2, \ldots, n$, where n the number of items counted so far.

The query, "*Find all departments with less than 7 employees*" can be expressed as follows:

```
count_emp(D#, mcount⟨E#⟩) ← emp(E#, Sal, D#).
large_dept(D#) ←            count_emp(D#, Count), Count = 7.
small_dept(D#, Dname) ←     dept(D#, Dname), ¬large_dept(D#).
```

This example illustrates some of the benefits of online aggregation. Negated queries are subject to existential variable optimization; thus, in $\mathcal{LDL}$++ the search for new employees of a department stops as soon as the threshold of 7 is reached. But the traditional `count` must retrieve all employees in the department, no matter how high their count is.

Several authors have advocated extensions to predicate calculus with generalized existential quantifiers [13,14], to express a concept such as "*There exist at least seven employees*". This idea is naturally supported by new aggregate `atleast⟨(K, X)⟩` that returns the value `yes` as soon as `K` instances of `X` are counted. This aggregate of Boolean behavior can be defined as follows:

```
single(atleast, (K, Y), 1).
multi(atleast, (K, Y), Old, New) ←   Old < K, New = Old + 1.
ereturn(atleast, (K, Y), K1, yes) ← K1 = nil, K = 1.
ereturn(atleast, (K, Y), K1, yes) ← K1 ≠ nil, K1 + 1 = K.
```

Then, a predicate equivalent to the `large_dept(D#)` can be formulated as follows:

```
lrg_dpt(D#, atleast⟨(7, Ename)⟩) ← empl(Ename, Sal, D#).
```

Here, because of the condition `Old < K` in the `multi` rule defining `atleast`, the search stops after seven employees, even for a positive goal ?`lrg_dpt(D#, yes)`, for which no existential optimization is performed.

Observe that `mcount` and `atleast` aggregates define monotonic mappings with respect to set containment: in the next section, we prove that all UDAs defined with only early returns (e.g., online aggregates) are monotonic, and can be freely used in recursive queries and rules.

7 Formal Semantics and Monotonicity

The logic-based semantics of a program with aggregates can be defined by viewing it as a short-hand of another Datalog program without aggregates. For that, we need the ability of enumerating the elements of the set one-by-one. For instance, if we assumed that the set elements belong to a totally ordered domain, then we could visit them one-at-a-time in, say, ascending order. But such an assumption would violate the genericity principle [1]; moreover, it still requires nonmonotonic constructs to visit the elements one-by-one, thus preventing the use of aggregates in recursive rules. A better solution consists in using `choice` [21,22], or more precisely the dynamic version of choice [12], which can used freely in recursive rules. By enforcing functional dependencies on the result produced by the rules, this powerful construct allows us to derive the Ordering rules, below, which arrange the elements of the set in a simple chain.

Positive choice programs are equivalent to programs with negated goals; these programs are guaranteed to have one or more total stable models [11]. As shown in [12], choice is strictly more powerful than other nondeterministic construct previously defined, including the *witness* operator of Abiteboul&Vianu [1], and the static version of choice [12]. This added power allows us to express computations that would not have been possible using witness or static choice. In particular a positive choice program can be used to order the elements of a sets into a chain [12]. This operation is critical in our inductive definition of aggregates discussed next.

For instance, say that we have the following rule where we apply `myagr` on the `Y`-values grouped by `X`:

$$\mathtt{r : p(X, myagr\langle Y\rangle) \leftarrow q(X, Y).}$$

The mapping from the body to head of this rule can expressed by (i) the $\mathtt{next_r}$ rules that arrange the `Y`-values of `q(Y)` into a chain, (ii) the `cagr` rules that implement the inductive definition of the aggregate by calling the `single` and `multi` rules, and (iii) the yield-rules that produce the actual pairs in `p` by using the `ereturn` and `freturn` rules.

The $\mathtt{next_{r1}}$ rules use the choice construct of $\mathcal{LDL}$++:

<ins>Ordering Rules:</ins>

```
next_r(X, nil, nil) ← q(X, Y).
next_r(X, Y1, Y2) ←   next_r(X, _, Y1), q(X, Y2),
                      choice((X, Y1), (Y2)), choice((X, Y2), (Y1)).
```

Aggregates can be defined by the following internal recursive predicate `cagr`:
<u>`cagr Rules`</u>

$$\begin{array}{ll}\texttt{cagr(myagr, X, Y, New)} \leftarrow & \texttt{next\_r(X, nil, Y), Y} \neq \texttt{nil,}\\ & \texttt{single(myagr, Y, New).}\\ \texttt{cagr(myagr, X, Y2, New)} \leftarrow & \texttt{next\_r(X, Y1, Y2),}\\ & \texttt{cagr(myagr, X, Y1, Old),}\\ & \texttt{multi(myagr, Y2, Old, New).}\end{array}$$

The `cagr` rules implement the inductive definition of the UDA by calling on the `single` and `multi` predicates written by the user. Therefore, `single` is used once to initialize `cagr(myagr, X, Y, New)`, where `Y` denotes the first input value and `New` is value of the aggregate on a singleton set. Then, for each new input value, `Y2`, and `Old` (denoting the last partial value of the aggregate) are fed to the `multi` predicate, to be processed by the `multi` rules defined by the user and returned to head of the recursive `cagr` rule.

Here, we have left the bodies of these rules unspecified, since no "special" restriction applies to them (except that they cannot use the predicate `p` being defined via the aggregate, nor any predicate mutually recursive with `p`).

The predicates `ereturn` and `freturn` are called by the yield rules that control what is to be returned:
<u>Early-Yield Rule</u>:

$$\begin{array}{ll}\texttt{p(X, AgrVal)} \leftarrow & \texttt{next\_r(X, nil, Y), Y} \neq \texttt{nil,}\\ & \texttt{ereturn(myagr, Y, nil, AgrVal).}\\ \texttt{p(X, AgrVal)} \leftarrow & \texttt{next\_r(X, Y1, Y2),}\\ & \texttt{cagr(myagr, X, Y1, Old),}\\ & \texttt{ereturn(myagr, Y2, Old, AgrVal).}\end{array}$$

The first early-yield rule applies to the first value in the set, and the second one to all successive values. The result(s) returned when all elements in the set have been visited is controlled by a final-yield rule:
<u>Final-Yield Rule:</u>

$$\begin{array}{ll}\texttt{p(X, AgrVal)} \leftarrow & \texttt{next\_r(X, \_, Y),} \neg\texttt{next\_r(X, Y, \_),}\\ & \texttt{cagr(myagr, X, Y, Old),}\\ & \texttt{freturn(myagr, Y, Old, AgrVal).}\end{array}$$

This general template defining the meaning of all aggregates is then customized by the user-supplied rules for `single`, `multi`, `ereturn`, and `freturn`, which all have `mvavg` as their first head argument (thus the aggregate name is used to avoid interference with other UDAs).

Monotonicity Observe that *negation is only used in the final yield rule.* When the aggregate definition contains no final-return rule (i.e., only early return rules) then the final-yield rule can be eliminated and the remaining rules constitute a *positive choice program*. Now, positive choice programs *define monotonic transformations*—an important result obtained in [12] that will be summarized next.

As customary in deductive databases, a program P can be viewed as consisting of two separate components: an extensional component, denoted $edb(P)$, and an intensional one, denoted $idb(P)$. Then, a positive choice program defines a monotonic multi-valued mapping from $edb(P)$ to $idb(P)$, as per the following theorem proven in [12]:

Theorem 1. *Let P and P' be two positive choice programs where $idb(P') = idb(P)$ and $edb(P') \supseteq edb(P)$. Then, if M is a choice model for P, then, there exists a choice model M' for P' such that $M' \supseteq M$.*

Thus, for a multi-valued function we only require that, as the value of the argument increases, some of the values of the function also increase (we do not require all values to increase). Furthermore, we say that we have a fixpoint when one of the function values is equal to its argument. Each multi-valued mapping also induces a nondeterministic (single-valued) mapping, defined as an arbitrary choice among the values of the function. As shown in [12], for choice programs, a fixpoint is reached by the inflationary repeated application of such a nondeterministic mapping.

Furthermore the set of these fixpoints coincide with the stable models [11] of the program obtained by rewriting the choice program into an equivalent program with negation [21]. Thus, our $\texttt{next}_\texttt{r}$ rules are formally defined by their equivalent rules with negated goals:

$$
\begin{array}{ll}
\texttt{next\_r}(\texttt{X},\texttt{Y1},\texttt{Y2}) \leftarrow & \texttt{next\_r}(\texttt{X},\_,\texttt{Y1}),\texttt{q}(\texttt{X},\texttt{Y2}),\\
 & \texttt{chosen}(\texttt{X},\texttt{Y1},\texttt{Y2}).\\
\texttt{chosen}(\texttt{X},\texttt{Y1},\texttt{Y2}) \leftarrow & \texttt{next\_r}(\texttt{X},\texttt{Y1},\texttt{Y2}),\neg\texttt{diffchoice}(\texttt{X},\texttt{Y1},\texttt{Y2}).\\
\texttt{diffchoice}(\texttt{X},\texttt{Y1},\texttt{Y2}) \leftarrow & \texttt{chosen}(\texttt{X},\texttt{Y1},\texttt{Y2}'),\texttt{Y2}' \neq \texttt{Y2}.\\
\texttt{diffchoice}(\texttt{X},\texttt{Y1},\texttt{Y2}) \leftarrow & \texttt{chosen}(\texttt{X},\texttt{Y1}',\texttt{Y2}),\texttt{Y1}' \neq \texttt{Y1}.
\end{array}
$$

This program, as every choice program reexpressed via negation, has one or more (total) stable models [11], where each stable model satisfies all the FDs defined by the choice goals [22,12].

Therefore, keeping with previous authors [16], we have defined the semantics of aggregates in terms of stable models [11]; however, through the use of the choice construct, we have avoided the computational intractability problems of stable models. Furthermore in our semantics, choice rules are only used to deliver the `next` value `Y2` generated by our rule r (for the given group-by value `X` and the previous such value `Y1`): thus, an operational realization is very simple and efficient since it reduces to a get-next operation on data. Furthermore, since aggregates define monotonic transformations in the usual lattice of set containment, bottom-up execution techniques of deductive databases, such as the semi-naive fixpoint, and magic sets, remain valid for these programs. Thus, monotone aggregates can be added to deductive database systems with no change in execution strategy—a conclusion that also applies to recursive queries with monotone aggregates in SQL DBMSs.

8 Programs with Monotone Aggregation

We now express several examples derived from [19] using our new monotonic aggregates [28].

Join the Party . The SQL-AG query of Example 4, can be expressed in $\mathcal{LDL}$ using the monotonic aggregate `mcount` and an additional predicate $\mathtt{c_friends}$.

$$\begin{array}{ll}
\mathtt{willcome(P)} \leftarrow & \mathtt{sure(P)}. \\
\mathtt{willcome(P)} \leftarrow & \mathtt{c\_friends(P, K), K >= 3}. \\
\mathtt{c\_friends(P, mcount}\langle\mathtt{F}\rangle) \leftarrow & \mathtt{willcome(F), friend(P, F)}.
\end{array}$$

Here, we have set $K = 3$ as the number of friends required for a person to come to the party. Consider now a computation of these rules on the following database.

```
sure(mark).     friend(jerry, mark).
sure(tom).      friend(penny, mark).
sure(jane).     friend(jerry, jane).
                friend(penny, jane).
                friend(jerry, penny).
                friend(penny, tom).
```

Then, the basic semi-naive computation yields:

$$\mathtt{willcome(mark), willcome(tom), willcome(jane),}$$

$$\mathtt{c\_friends(jerry, 1), c\_friends(penny, 1), c\_friends(jerry, 2),}$$

$$\mathtt{c\_friends(penny, 2), c\_friends(penny, 3), willcome(penny),}$$

$$\mathtt{c\_friends(jerry, 3), willcome(jerry).}$$

This example illustrates how the standard semi-naive computation can be applied to queries containing monotone user-defined aggregates.

The Join-the-Party query of Example 4 eliminates the need for a `c_friends` predicate by using the 'having' construct . In $\mathcal{LDL}$++, we can obtain the same effect by using the aggregate `atleast` defined in Section 6, which is also monotone:

$$\begin{array}{ll}
\mathtt{wllcm(F, yes)} \leftarrow & \mathtt{sure(F)}. \\
\mathtt{wllcm(X, atleast}\langle(\mathtt{3, F})\rangle) \leftarrow & \mathtt{wllcm(F, yes), friend(X, F)}.
\end{array}$$

Unlike in the previous formulation, where a new tuple `c_friends` is produced every time a new `friend` is found, a new `wllcm` tuple is here produced only when the threshold of 3 is crossed.

Company Control Another interesting example is transitive ownership and control of corporations. Say that owns(C1, C2, Per) denotes the percentage of shares that corporation C1 owns of corporation C2. Then, C1 controls C2 if it owns more than, say, 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 predicate defined as follows:

```
control(C, C) ←         owns(C, _, _).
control(Onr, C) ←       towns(Onr, C, Per), Per > 50.
towns(Onr, C2, msum⟨Per⟩) ← control(Onr, C1), owns(C1, C2, Per).
```

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, towns computes transitive ownership with the help of msum that adds up the shares of controlling companies. Observe that any pair (Onr, C2) is added at most once to control, thus the contribution of C1 to Onr's transitive ownership of C2 is only accounted once.

9 Conclusion

The practical importance of database aggregates has long been recognized, but indepth treatments of this critical subject were lacking. In this paper, we have addressed both the theoretical and practical aspects of aggregates, including user-defined aggregates and online aggregation. Our logic-based formalization of aggregates provided a simple and practical solution to problem of monotone aggregation, a problem on which many previous approaches had achieved only limited success [17,15,10,19].

Various examples were also given illustrating power and flexibility of UDAs in advanced applications; several more examples, omitted because of space limitations, can be found in [25]. For instance, by adding greedy aggregates built upon priority queues, we expressed graph algorithms such as Dijkstra's single source least-cost path, or Prim's least-cost spanning tree. Also data mining functions, including tree classifiers and A Priori, can be formulated efficiently using our UDAs.

At UCLA, we developed the the SQL-AG prototype that supports the UDAs here described on top of DB2 [25], and we also developed a new version of $\mathcal{LDL}$++ [28] supporting the Datalog extensions described in this paper. The SQL-AG implementation is of particular significance, since it shows that UDAs are fully compatible with O-R systems, and can actually outperform builtin aggregates in particular applications.

We are currently investigating the issue of ease of use in UDAs. In fact, while UDAs in $\mathcal{LDL}$++ can be expressed using rules, several procedural language functions must be written to add a new UDA in SQL-AG or SQL3. However, our experience suggests that in most UDAs the computations to be performed by the INITIALIZE, ITERATE, TERMINATE, and PRODUCE functions are very simple, and can effectively be expressed using an (SQL-like) high-level language. We expect that this approach will enhance users' convenience, and portability. A simple SQL-like language for UDAs is described in [27].

References

1. Abiteboul S., Hull R. and Vianu V., Foundations of Databases, Addison Wesley, 1995. 54
2. S. Brin, R. Motwani, J. D. Ullman, S. L. Tsur, "Dynamic Itemset Counting and Implication Rules for Market Basket Data". In *SIGMOD'97*. 48
3. D. Chamberlin, "Using the new DB2, IBM's Object-Relational Database System," Morgan Kaufmann, 1996. 45, 49
4. Cindy Xinmin Chen, Carlo Zaniolo: Universal Temporal Extensions for Database Languages. ICDE 1999: 428-437. 45, 46
5. Charles Elkan. "Boosting and Naive Bayesian Learning". Technical report no cs97-557, Dept. of Computer Science and Engineering, UCSD, September 1997. 44
6. J. M. Hellerstein, P. J. Haas, H. J. Wang. "Online Aggregation". *SIGMOD, 1997*. 46
7. M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, J. D. Ullman, "Computing Iceberg Queries Efficiently". In *VLDB 1998*. 48
8. ISO/IEC JTC1/SC21 N10489, ISO//IEC 9075, "Committee Draft (CD), Database Language SQL", July 1996. 43, 45, 46
9. Finkelstein, S. J., N. Mattos, I. S. Mumick, and H. Pirahesh, Expressing Recursive Queries in SQL, ISO WG3 report X3H2-96-075, March 1966. 47
10. S. Ganguly, S. Greco, and C. Zaniolo, "Extrema Predicates in Deductive Databases," JCSS 51(2): 244-259 (1995). 47, 50, 58
11. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. *Procs. Joint International Conference and Symposium on Logic Programming*, pp. 1070–1080, 1988. 48, 54, 56
12. F. Giannotti, D. Pedreschi, and C. Zaniolo, "Semantics and Expressive Power of Non-Deterministic Constructs in Deductive Databases," JCSS, to appear. 54, 56
13. Gyssen, M., Van Gucht, D. and Badia, A., Query Languages with Generalized Quantifiers, in Applications of Logic Databases, R. Ramakrishan, Kluwer, 1995. 53
14. Hsu, P. Y. and Parker, D. S., "Improving SQL with Generalized Quantifiers," Proc. ICDE 1995. 53
15. A. Van Gelder. "Foundations of Aggregations in Deductive Databases." *Proc. of the Int. Conf. On Deductive and Object-Oriented databases*, 1993. 47, 50, 58
16. David B. Kemp and Peter J. Stuckey, "Semantics of logic programs with aggregates" *Proc. 1991 International Symposium on Logic Programming*, pages 387–401, October 1991. 56
17. D. Srivastava, R. Ramakrishnan, P. Seshadri, S. Sudarshan. Coral++: Adding Object-Orientation to a Logic Database Language. In *VLDB 1993: 158-170*. 47, 53, 58
18. R. Rastogi, K. Shim. "PUBLIC: A Decision Tree Classifier that Integrates Building and Pruning". VLDB 1998: 404-415. 48
19. K. A. Ross and Yehoshua Sagiv, "Monotonic Aggregation in Deductive Database", JCSS 54(1), 79-97 (1997). 47, 50, 57, 58

20. S. A. Naqvi, S. Tsur *"A Logical Language for Data and Knowledge Bases"*, W. H. Freeman, 1989. 51
21. D. Saccà and C. Zaniolo. Stable models and non-determinism in logic programs with negation, *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 205–217, 1990. 54, 56
22. D. Saccà and C. Zaniolo, Deterministic and non-deterministic Stable Models, *Journal of Logic and Computation*, 7(5):555-579, October 1997. 54, 56
23. S. Sarawagi, S. Thomas, R. Agrawal, "Integrating Association Rule Mining with Relational Database Systems: Alternatives and Implications". In *SIGMOD, 1998*. 45
24. J. C. Shafer, R. Agrawal, M. Mehta, "SPRINT: A Scalable Parallel Classifier for Data Mining". In *VLDB 1996*. 45
25. Haixun Wang, The SQL-AG System, `http://magna.cs.ucla.edu/~hxwang/sqlag/sqlag.html` 49, 58
26. H. Wang and C. Zaniolo "User-Defined Aggregates in Datamining," ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, DMKD'99, May 30, 1999. 46
27. H. Wang, and C. Zaniolo "User Defined Aggregates in Object-Relational Systems" ICDE 2000 Conference, San Diego, CA, February 29-March 3, 2000. 59
28. C. Zaniolo et al. $\mathcal{LDL}$++ Documentation and Web Demo, `http://www.cs.ucla.edu/ldl` 51, 53, 57, 58