**Krzysztof R. Apt, Victor W. Marek,
Mirosław Truszczyński and
David S. Warren (Eds.)**

The Logic Programming Paradigm: a 25-Year Perspective

Springer

Berlin Heidelberg New York
Barcelona Budapest Hong Kong
London Milan Paris
Santa Clara Singapore Tokyo

Logic-Based User-Defined Aggregates for the Next Generation of Database Systems

Carlo Zaniolo and Haixun Wang

Computer Science Department, University of California at Los Angeles, Los Angeles, CA 90095, USA

Summary. In this paper, we provide logic-based foundations for the extended aggregate constructs required by advanced database applications. In particular, we focus on data mining applications and show that they require user-defined aggregates extended with early returns. Thus, we propose a simple formalization of extended user-defined aggregates using the nondeterministic construct of choice. We obtain programs that have a formal semantics based on the concept of total stable models, but are also amenable to efficient implementation. Our formalization leads to a simple syntactic characterization of user-defined aggregates that are monotone with respect to set containment. Therefore, these aggregates can be freely used in recursive programs, and the fixpoints for such programs can be computed efficiently using the standard techniques of deductive databases. We describe the many new applications of user-defined aggregates, and their implementation for the logical data language $\mathcal{LDL}$++. Finally, we discuss the transfer of this technology to SQL databases.

1 Introduction

A new wave of database applications, particularly decision-support and data mining applications, are based on complex aggregates not supported by current DBMSs: in fact, SQL2 specifications only prescribe support for the five built-in aggregates of *sum, count, avg, max and min.* To remedy this problem, many vendors have moved to add new aggregates, such as data cubes [12] in their commercial releases. However, ad-hoc extensions by vendors are never enough, since data mining applications alone require a wide variety of new aggregate functions; moreover, other application domains, such as temporal reasoning and time-series analysis, require many new aggregates as well. A general solution consists in extending Database Management Systems (DBMSs) with *user-defined aggregates* (UDAs)—a functionality already provided, in Postgres [24] and SQL3 [19]. As we shall discuss later in the paper, however, the approaches followed by SQL3 and Postgres suffer from various limitations, which can be summarized as follows:

- Lack of declarative semantics, since the aggregates are defined via functions coded in a procedural language, and
- the aggregates so defined cannot express many of the aggregates used by data-mining applications (e.g., online aggregation), and cannot be used

in defining recursive predicates because of their nonmonotonic nature (in SQL3, negation and aggregates can only be used in stratified queries).

In this paper, we provide a simple and general solution to these problems as follows. In the next section, we discuss the different kinds aggregations required by new applications. In Section 3, we focus on the problems that current aggregates have in supporting these applications and, in Section 4, we propose user-defined aggregates for Datalog that express all the aggregates of interest. We also describe their efficient implementation in the extended $\mathcal{LDL}$++ system implemented at UCLA (Section 6). In Section 5, we propose a formal logic-based semantics for aggregates using the notion of stable models and nondeterministic choice. This semantics leads to a simple syntactic characterization of monotone aggregates, Section 7. Monotone aggregates can thus be used to support new applications, which are discussed in Section 8. The significance of these advances for SQL DBMSs is discussed in section 9.

2 New Applications Require New Aggregates

Our study of data mining and decision support applications suggest that data mining algorithms

- make heavy use of aggregates, but
- often employ them in forms that cannot be supported well or efficiently by SQL2.

Similar conclusions also apply to other advanced applications; however, data mining and decision support provide the best evidence of this general trend. As a first example, consider the data mining methods used for classification. For instance, say that we want to classify the value of PlayTennis as a 'Yes' or a 'No' given the following vector of attribute values

Outlook, Temperature, Humidity, Wind.

and a training set such as that shown in Table 1.

The algorithm known as Boosted Bayesian Classifier [10] 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* [10] 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:

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

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

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 counts appearing in the numerator and denominator, above, can be easily computed using SQL aggregate queries. For instance, all the numerators values for the third column (the Wind column) can can be computed as follows:

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

For the training set of Table 1 this query returns:

```
(Weak, Yes, 6)
(Weak, No, 2)
(Strong, Yes, 3)
(Strong, No, 3)
```

Similar aggregate queries on the other three columns compute the remaining coefficients for the numerator, while the denominator can be evaluated by counting tuple grouped by PlayTennis. Then, the boosting step described in [29] can be implemented by simply increasing the weight of the misclassified tuples. In conclusion, this award-winning classification algorithm can be implemented well using the SQL `count` aggregate. This aggregate-based formulation is preferable to the original formulation given in [29] since it ensures scalability and performance on large training sets.

Count-like aggregates are also the linchpin of other classifiers, such as the one in [2] that will be discussed briefly in Section 8.

Associations Many applications designed to discover associations make extensive use of count-like aggregates. For instance, consider the following example:

$$\mathsf{citizen(x,c)},\ \mathsf{official\_language(c,l)} \Rightarrow \mathsf{speaks(x,l)}$$

If citizen, official_language and speaks are stored as database relations, respectively named C, O and S, then the our confidence in the validity of this rule can be estimated as follows:

$$C_f = \frac{|(\pi_{1,3}C \bowtie O) \cap S|}{|(\pi_{1,3}C \bowtie O|} \tag{1}$$

The closer C_f is to 1, the stronger is our confidence in the validity of the rule. C_f can be computed by SQL queries using count.

In [29] we assume that we are given a database of relations and we describe a system to find implication rules, such as the one above, by searching for tables that satisfy some meta-level templates specified by the user. For instance, the transitive composition template applicable to our previous example is:

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

where P, Q and R now stand for arbitrary database predicates. Then, the system search for the triplets P, Q and R of database relations for which C_f exceeds a given threshold, e.g., satisfies the condition $C_f > 0.8$. In [29], we described a system that uses the meta-level predicates of $\mathcal{LDL}$++ to perform such a search on a given database schema (using SQL's count aggregate on the underlying database).

In summary, some interesting association and classification methods can be implemented directly using the standard SQL2 aggregates. On the other hand, many complex and specialized aggregate computations used by other data mining algorithms cannot be expressed effectively using standard SQL2 aggregates. For instance, market data analysis represents one of the best known applications of association rules. The Apriori algorithm which is at core of this application searches for frequent items sets: i.e., it counts items and combination of items that are above a given threshold [2,3,5]. While this computation can be expressed using the SQL `count` aggregate, ensuring the efficiency of these queries represents much more of a challenge [27]. The same is true for a related class of queries called *iceberg queries* [11]. Also the classification algorithms discussed in [2] face similar problems—see discussion in Section 9.

In general, aggregate-like computations used in data mining algorithms cannot be implemented conveniently and efficiently using built-in SQL2 aggregates, or even employing SQL3's user-defined aggregates. We will now discuss their limitations and then propose an extended form of aggregate to overcome such limitations.

Online aggregation The recently proposed concept of online aggregation finds many applications in data mining and OLAP algorithms [17]. A typical use of online aggregation is for estimating averages. Assuming that there is no skewing in the way the data is stored (or it is retrieved in ways that avoid skewing), the computation converges toward the final value after a relatively small portion of a large dataset is visited. Online aggregation provides a simple solution to this problem by the mechanism of *early returns*. By returning partial results, the online version of the aggregate allows the user to stop the computation as soon as convergence is reached within the desired degree of accuracy.

There are many situations where online aggregation is highly desirable; for instance, we can use it to speed-up the discovery of association rules, since C_f can be computed as the average value of a function f defined as follows: for each tuple contained in the denominator of Equation 1, f evaluates to one if the tuple is also contained in the numerator, and evaluates to zero otherwise. Thus, C_f can be evaluated using the following $\mathcal{LDL}$++ rule, where $\texttt{avg}\langle\texttt{W}\rangle$ in the head denotes that the aggregate `avg` is computed on the set of `W`-values satisfying the goals in the body of the rule:

$$\begin{aligned}\texttt{overlap}(\texttt{avg}\langle\texttt{W}\rangle) \leftarrow\ & \texttt{citizen}(\texttt{X},\texttt{Ct}), \texttt{official\_language}(\texttt{Ct},\texttt{L}),\\ & \texttt{if}(\texttt{speaks}(\texttt{X},\texttt{L})\ \textbf{then}\ \texttt{W} = 1\ \textbf{else}\ \texttt{W} = 0).\end{aligned}$$

Here, online aggregation can speed up the search dramatically, since the evaluation of most relation triplets can be dropped early if online average estimates are seen that fall well below the minimum threshold.

The ability of a computation to produce "early returns" is useful in many other situations besides online aggregation. For instance, to perform time-series analysis we need many different sorts of temporal aggregation, such as cumulative aggregation, and moving-window aggregation[21]. Thus, a user might request the running sum of sales, i.e., the running sum of the sales from a given time. This computation is facilitated by the fact that the data is normally stored sorted by time, as in Table 2. Then, for each new sale, the running sum simply adds the new amount to the running value and returns it as a partial result while continuing this one-pass computation. Thus the computation of a running sum is simple to express using the early-return mechanism of online aggregation. In SQL2, however, the running sum must be expressed as the sum of all sales up to the time of a given sale—a separate computation for each sale. SQL2 will also perform poorly in the computation of moving-window aggregate, where, e.g., we want to compute the average of last n sales, or the average of sales that took place during the last m days.

Year	Season	Month	Day	Year-Sales,	Season-Sales,	Month-Sales,	Day-Sales
1992	winter	feb	6	?	?	?	409
1992	winter	feb	26	?	648	648	239
1992	spring	mar	7	?	318	318	318
1992	summer	jun	23	?	?	394	394
1992	summer	jul	1	?	?	167	167
1992	summer	aug	22	?	?	?	176
1992	summer	aug	24	?	993	432	256
1992	fall	oct	13	?	?	?	481
1992	fall	oct	21	?	?	597	116
1992	fall	nov	11	2692	733	136	136

Table 2. Keeping track of Sales over Time

Another operation frequently used in OLAP and ROLAP applications is the computation of DataCubes [12]. A datacube is computed by *rolling-up* an aggregate over the different dimensions of data. In the temporal dimension, for instance, we might roll-up the sales by Year, Season, Month, and Day. In SQL2, we could find the daily sales by using group by Year, Season, Month, Day. and then find the monthly sales by group by Year, Season, Month, and so. This is quite wasteful since the computation of the monthly sales can be done in one pass over data that is temporally ordered. Therefore, many commercial DBMSs have now added special extensions for roll-up aggregates. However, it is easy to see that an early-return construct that supports online aggregation can also be used to compute roll-ups for data cubes. In fact, when the input changes from a day (month) to the next day (month), the aggregate value for the old day (month) can be returned while the aggregate computation for the new day (month) is restarted anew.

In summary, the ability of supporting early returns represents a critical extension that gives user-defined aggregates the ability to support a new wave of data-intensive applications, particularly data-mining applications. In the rest of the paper we develop the logic-based foundations for user-defined aggregates with and without early returns.

3 User-Defined Aggregates: the State of the Art

Postgres and SQL3 Postgres allows the user to specify aggregates in terms of state-transition functions [24,30]. The state functions sfunc1 and sfunc2 compute the new state from the old state and the value of the new record. Finally, initcond is used to define the initial state of the aggregate, while finalfunc defines the computation to be performed at the end.

Postgres' idea of defining new aggregates by giving the functions for the initial state, the iteration state, and the final state has also influenced the SQL3 proposal. SQL3 prescribes that *iterative routines* should be used to define user-defined aggregates in some external language. In SQL3 [19], iterative routines consist of three parts:

- a *initialize routine* that gives an initial value to the aggregate
- an *iterate routine* yielding a new value for each new record in the table
- a *terminate routine* that returns the final value computed for the aggregate.

The $\mathcal{LDL}/\mathcal{LDL}$++ System The logical data language $\mathcal{LDL}$++, developed at MCC [9,33] supports user-defined aggregates. In $\mathcal{LDL}$++, an aggregate function f on a set S, can be introduced via the following inductive definition:

1. *Base*: the value $f(\{x\})$ on a singleton set must be defined in terms of x,
2. *Induction*: for any other set $S' = S \sqcup \{x\}$ the value of $f(S')$ must be defined in terms of $f(S)$ and x.

For instance, count and sum can be expressed as follows:

$Count$: $\mathsf{f}(\{\mathsf{x}\}) = 1$ *and* $\mathsf{f}(\mathsf{S}\sqcup\{\mathsf{x}\}) = \mathsf{f}(\mathsf{S}) + 1$
Sum : $\mathsf{f}(\{\mathsf{x}\}) = \mathsf{x}$ *and* $\mathsf{f}(\mathsf{S}\sqcup\{\mathsf{x}\}) = \mathsf{f}(\mathsf{S}) + \mathsf{x}$

In $\mathcal{LDL}$++, the computations pertaining to the base step and the induction step are expressed using, respectively, the predicates `single` and `multi`. Thus, `count` and `sum` could be expressed using the following rules:

For `count`:

```
single(count, Y, 1).
multi(count, Y, Old, New) ← New = Old + 1.
```

For `sum`:

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

Once defined, the aggregates can be applied to concrete sets, or to virtual sets defined in the heads of rules. In this paper, we concentrate on the second situation, which syntactically, is denoted by pointed bracket pairs '$\langle\ldots\rangle$' in the head of a rule. For instance, to compute the number of employees in each department, we can write:

```
dept_size(D#, count⟨E#⟩) ← empl(E#, Sal, D#).
```

Here, ⟨`E#`⟩ denotes the set of `E#` *grouped by* the remaining columns in the head (i.e., grouped by `D#`). Thus `count⟨E#⟩` counts the employees for each department.

$\mathcal{LDL}$++ rule-based framework makes it easy to define new aggregates. For instance, to find the highest paid employees in each department along with their salaries we can write:

$$\texttt{top\_paid(D\#, maxpair}\langle\texttt{(E\#, Sal)}\rangle\texttt{)} \leftarrow \texttt{emp(E\#, Sal, D\#)}.$$

Here, `maxpair` denotes a new aggregate defined over sets of pairs (`E#, Sal`) grouped by `D#`. Now, `maxpair` can be defined by the user using the following rules:

```
single(maxpair, (E#, Sal), (E#, Sal)).
multi(maxpair, (E#, Sal), (Eold, Sold), (E#, Sal)) ←      Sal >= Sold.
multi(maxpair, (E#, Sal), (Eold, Sold), (Eold, Sold)) ← Sold >= Sal.
```

Observe that the two `multi` rules used to extract the maxima are not mutually exclusive. Thus, when a new employee is found with salary equal to the current max, both the new and old `E#, Sal` pairs are accepted as potential answers by our `maxpair` aggregate. Once defined, an aggregate can be used as any built-in aggregate. Thus, to derive the department with the max count of employees, we write:

$$\texttt{largest\_dept(maxpair}\langle\texttt{(D\#, Ecount)}\rangle\texttt{)} \leftarrow \texttt{dep\_size(D\#, Ecount)}.$$

Thus, in the application we want to preserve ties and return them in the result. Consider however the situation of a greedy algorithm, such as Prim's algorithm that generates a least cost spanning tree [16]. At each step of the computation, a least cost arc is added to the spanning tree being constructed—i.e., if several arcs share the same cost only one can be used—otherwise the final result is no longer a tree. This tie-breaking version of max-pair (actually of its dual min-pair), can be specified by simply replacing the ">=" condition in one of the two multi-maxpair rules above by ">". Thus the particular result returned by the aggregate depends on the order in which the the input data is processed (non-determinism).

These examples illustrate some of the properties needed in user-defined aggregates.

- Aggregates often take multiple fields (i.e., structured records) as an input, and return multiple fields for each record returned
- Aggregates often need to return multiple records (e.g., several employees sharing the same top salary in the department)
- Aggregates often return different results depending on the order in which the data is visited.

The newly proposed SQL3 standards, and the original version of $\mathcal{LDL}$++ developed at MCC, meet these basic requirements for user-defined aggregates; however they do not support early returns.

4 Aggregates with Early Returns

SQL3, Postgres, and the original version of $\mathcal{LDL}$++ developed at MCC cannot express important classes of aggregates, such as online aggregation or roll-ups for data cubing. This is due to the fact they not support early returns, since they assume that results are only produced at end of the computation. To remedy this limitation, at UCLA we have developed extensions of $\mathcal{LDL}$++ and SQL to support *early returns*, whereby partial results are produced during the computation of user-defined aggregates.

The extension of $\mathcal{LDL}$++ developed at UCLA supports the predicate `ereturn` along with `single` and `multi` previously discussed. Now, the user can define new online aggregates by simply writing rules for the situations where partial results must be returned during the on-going calculation of the aggregate. For instance, the following computation of averages produces a partial result every fifth element.

$$
\begin{array}{ll}
\texttt{single}(\texttt{onl\_avg}, \texttt{X}, (\texttt{X}, \texttt{1})). & \\
\texttt{multi}(\texttt{onl\_avg}, \texttt{X}, (\texttt{Sum}, \texttt{Count}), (\texttt{Sum} + \texttt{X}, \texttt{Count} + \texttt{1})). & \\
\texttt{ereturn}(\texttt{onl\_avg}, \texttt{X}, (\texttt{Sum}, \texttt{Count}), \texttt{Avg}) & \leftarrow \texttt{Count mod 5} = \texttt{0}, \\
 & \quad \texttt{Avg} = \texttt{Sum}/\texttt{Count}.
\end{array}
$$

The new version of $\mathcal{LDL}$++ also gives the user explicit control of final returns, using the predicate `freturn`. For instance to produce a final return for our `onl_avg`, without producing the same values as early returns, the user might write:

$$
\begin{array}{ll}
\texttt{freturn}(\texttt{onl\_avg}, \texttt{X}, (\texttt{Sum}, \texttt{Count}), \texttt{Avg}) & \leftarrow \texttt{Count mod 5} \neq \texttt{0}, \\
 & \quad \texttt{Avg} = \texttt{Sum}/\texttt{Count}.
\end{array}
$$

The new version of $\mathcal{LDL}$++ is upward-compatible with the old one: when the return rules are omitted the results are only returned at the end of the aggregate computation.

Although the early-return capability was specifically introduced to support online aggregation, we found that it enabled us to define a large number of new aggregates which would have been impossible (or prohibitively difficult and inefficient) to express without it. Take for instance our previous roll-ups example of Table 2. It is simple to code this by using the return rules to detect the changes in $(\texttt{Year}, \ldots, \texttt{Day})$ and returning the accumulated result for the columns which have changed and a question mark for the other columns. In fact, using user-defined aggregates extended with early returns, we implemented an assortment of complex aggregations including: online queries, median estimation, roll-up aggregates for data cubing, various temporal aggregates for time series, set-comparison aggregates, and Allen temporal operators [6]. Because of space limitation, we refer the reader to the detailed discussion, and live demo, in `http://www.cs.ucla.edu/ldl`.

We conjecture that user-defined aggregates so extended can support all aggregates that can naturally be computed in a single pass through the data. Aggregates, such as data cubes, where the computation requires several passes, can then be expressed by combining several user-defined aggregates.

The benefits of early returns extend beyond data mining applications. For instance, let us define a version of count, called `mcount` that returns the incremental count at each step.

$$
\begin{array}{ll}
\texttt{single(mcount, Y, 1).} & \\
\texttt{multi(mcount, Y, Old, New)} \leftarrow & \texttt{New} = \texttt{Old} + 1. \\
\texttt{ereturn(mcount, Y, Old, 1)} \leftarrow & \texttt{Old} = \texttt{nil}, \texttt{New} = 1 \\
\texttt{ereturn(mcount, Y, Old, New)} \leftarrow & \texttt{Old} = \texttt{nil}, \texttt{New} = \texttt{Old} + 1.
\end{array}
$$

The condition $\texttt{Old} = \texttt{nil}$ is only satisfied when the first item in the set is found. Thus the first `ereturn` rule produces a value of 1 for `mcount`. The second rule returns all the values, $2, ..., n$ where n the actual cardinality of the set. At a first impression this aggregate might appear, strange and inefficient, but consider an actual example:

For instance the query "*Find all departments with more than 7 employees*" can be expressed using the following pairs of $\mathcal{LDL}$++ rules:

$$
\begin{array}{ll}
\texttt{count\_emp(D\#, mcount}\langle\texttt{E\#}\rangle\texttt{)} \leftarrow & \texttt{emp(E\#, Sal, D\#).} \\
\texttt{large\_dept(D\#)} \leftarrow & \texttt{count\_emp(D\#, Count), Count} = 7. \\
\texttt{small\_dept(D\#, Dname)} \leftarrow & \texttt{dept(D\#, Dname)}, \neg\texttt{large\_dept(D\#).}
\end{array}
$$

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

Several authors have advocated extensions to predicate calculus with generalized existential quantifiers [18,14,32], to express a concept such as "*There exist at least seven employees*". This idea is naturally supported by new aggregate $\texttt{atleast}\langle(\texttt{K}, \texttt{X})\rangle$ that returns the value `yes` as soon as `K` `X`s are counted. This aggregate of Boolean behavior can be defined as follows:

$$
\begin{array}{ll}
\texttt{single(atleast, (K, Y), 1).} & \\
\texttt{multi(atleast, (K, Y), Old, New)} \leftarrow & \texttt{Old} < \texttt{K}, \texttt{New} = \texttt{Old} + 1. \\
\texttt{ereturn(atleast, (K, Y), K1, yes)} \leftarrow & \texttt{K1} = \texttt{nil}, \texttt{K} = 1. \\
\texttt{ereturn(atleast, (K, Y), K1, yes)} \leftarrow & \texttt{K1} \neq \texttt{nil}, \texttt{K1} + 1 = \texttt{K}.
\end{array}
$$

Then, the previous queries searching for departments with more than 7 employees can simply be expressed as follows:

$$
\texttt{large\_detp(D\#)} \leftarrow \texttt{count\_emp(D\#, atleast}\langle(3, \texttt{E\#})\rangle\texttt{)}
$$

Here, because of the condition $\texttt{Old} < \texttt{K}$ in the `multi` rule defining `atleast`, the search stops after seven employees, even for a positive goal, such as ?$\texttt{lrg\_dpt(D\#, yes)}$, for which no existential optimization is performed.

Temporal queries provide a particularly fertile ground for the use of the new aggregates as demonstrated by the temporal extension of Datalog described in [6]. Another related domain is time-series analysis, where various kinds of computations that can be performed in a single pass through the data stream can be expressed using our extended aggregates. For instance, moving-window aggregates represent a type of computation that is performed frequently. In the following example we have sequences of closing prices for stocks, and we want to average the prices of IBM stocks over the last five days. Then we write the following $\mathcal{LDL}$++ program for which we defined the `mw5avg` aggregate:

$$\texttt{p(mw5avg}\langle\texttt{A}\rangle\texttt{)} \leftarrow \texttt{stock\_closing('IBM', A).}$$

$$\begin{array}{ll}\texttt{single(mw5avg, Y, [Y]).} & \\ \texttt{multi(mw5avg, Y0, OL, NL)} \leftarrow & \texttt{if(OL = [Y1, Y2, Y3, Y4]} \\ & \texttt{then L = [Y1, Y2, Y3]} \\ & \texttt{else L = OL), NL = [Y0|L].}\end{array}$$

$$\begin{array}{l}\texttt{ereturn(mw5avg, Y0, [Y1, Y2, Y3, Y4], Avg)} \leftarrow \\ \qquad\qquad\qquad \texttt{Avg = (Y0 + Y1 + Y2 + Y3 + Y4)/5.}\end{array}$$

Therefore, the `return` rule, after five values have accumulated, returns their sum divided by 5; nothing is returned before that. The `multi` rule eliminates the least recent value from the list (but only after this has reached a length of five) and then adds the most recent value that becomes the head of the list.

5 Formal Semantics

The formal semantics of extended aggregates can be defined using the standard declarative semantics of Datalog, by simply viewing a program with aggregates as a short-hand for an expanded program without aggregates.

Take for instance, the following rule where we compute a new aggregate function `myagr` on a set of `Y`-values grouped by `X`:

$$r_1 : \texttt{p(X, myagr}\langle\texttt{Y}\rangle\texttt{)} \leftarrow \texttt{q(X, Y).}$$

The computation of a set aggregate requires the ability to enumerate the elements of the set one-by-one. For instance, if we make the assumption that the set elements are totally ordered, then we can visit them one-at-a-time in, say, ascending order. But such an assumption would violate the genericity principle [1], and still requires nonmonotonic constructs to visit the elements on-by-one, thus preventing the use of aggregates in recursive rules. A better solution consists in using `choice` [25,26] which can be freely used in recursive rules. In fact, positive programs with choice are equivalent to programs with negated goals which are guaranteed to have one or more total stable models [13].

Therefore, the first rewriting step for $\mathtt{r}_1$ uses choice to produce a *chain* of results, rather than a set of results, from the body of $\mathtt{r}_1$:

Ordering Rules:

$$
\begin{array}{ll}
\mathtt{next\_r_1(X, nil, nil)} \leftarrow & \mathtt{q(X, Y).} \\
\mathtt{next\_r_1(X, Y1, Y2)} \leftarrow & \mathtt{next\_r_1(X, \_, Y1), q(X, Y2),} \\
 & \mathtt{choice((X, Y1), (Y2)), choice((X, Y2), (Y1)).}
\end{array}
$$

Aggregates are computed by the following internal recursive predicate `cagr`:

`cagr` Rules

$$
\begin{array}{ll}
\mathtt{cagr(myagr, X, Y, New)} \leftarrow & \mathtt{next\_r_1(X, nil, Y), Y \neq nil,} \\
 & \mathtt{single(myagr, Y, New),} \\
\mathtt{cagr(myagr, X, Y2, New)} \leftarrow & \mathtt{next\_r_1(X, Y1, Y2), cagr(myagr, X, Y1, Old),} \\
 & \mathtt{multi(myagr, Y2, Old, New).}
\end{array}
$$

The recursive `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.

This general template defining aggregates is customized for each UDA by the user-supplied rules for `single`, `multi`, `ereturn` (for early returns), and `freturn` (for final returns):

Single, Multi, Early-Return, and Final-Return Rules

$$
\begin{array}{ll}
\mathtt{single(myagr, Y, New)} \leftarrow & \ldots \\
\mathtt{multi(myagr, Y, Old, New)} \leftarrow & \ldots \\
\mathtt{ereturn(myagr, Y, Old, Value)} \leftarrow & \ldots \\
\mathtt{freturn(myagr, Y, Old, Value)} \leftarrow & \ldots
\end{array}
$$

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:

Early-Yield Rule:

$$
\begin{array}{ll}
\mathtt{p(X, AgrVal)} \leftarrow & \mathtt{next\_r_1(X, nil, Y), Y \neq nil,} \\
 & \mathtt{ereturn(myagr, Y, nil, AgrVal).} \\
\mathtt{p(X, AgrVal)} \leftarrow & \mathtt{next\_r_1(X, Y1, Y2), cagr(myagr, X, Y1, Old),} \\
 & \mathtt{ereturn(myagr, Y2, Old, AgrVal).}
\end{array}
$$

The first early-yield rule applies to the first value in the set, and the second one to all successive values. The values returned once all the values in the set have been visited is controlled by a final-yield rule:
Final-Yield Rule:

$$\begin{aligned}\mathtt{p(X, AgrVal)} \leftarrow\ & \mathtt{next\_r_1(X, \_, Y)}, \neg\mathtt{next\_r_1(X, Y, \_)},\\ & \mathtt{cagr(myagr, X, Y, Old)},\\ & \mathtt{freturn(myagr, Y, Old, AgrVal)}.\end{aligned}$$

Therefore, the meaning of a program P containing a rule such as $\mathtt{r_1}$, is simply the program P' obtained by replacing $\mathtt{r_1}$ with the ordering rules, the cagr rules, the early-yield rules, and the final-yield rules, while leaving the remaining rules in P unchanged (including the user-supplied single, multi, early-return, and final-return rules). Now, *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*; positive choice programs *define monotonic transformations* [15]. The monotonicity property is discussed in more details in the next section.

In summary, our powerful user-defined predicates are simply a syntactic short-hand requiring no semantic extensions to an enhanced Datalog language such as $\mathcal{LDL}$++. The use of special syntactic constructs, however, allows their direct implementation to achieve significant performance gains.

6 Monotonic Aggregation

The semantics of choice can be formally defined using the notion of stable model [13] . In fact, each program with choice can be rewritten into an equivalent program with negation which is guaranteed to have one or more total stable models [25]. Multiple canonical models tantamount to nondeterminism. Indeed, choice inherits the nondeterministic aspects of stable models [26], without suffering from any of the computational intractability problems of stable models. Programs with choice can be computed efficiently since choice defines a monotone transformation. For instance, consider the rule:

$$r_2 : \mathtt{p(X, Y)} \leftarrow \mathtt{q(X, Y)}, \mathtt{choice((X), (Y))}.$$

If P and Q respectively denote a set of p-atoms and q-atoms, the transformation defined by r_2 consists in selecting a maximal subset $P \subseteq Q$ such that the FD $X \rightarrow Y$ holds in P. This transformation is multivalued i.e., in general, more than one such maximal subset exits and, therefore, one must be selected arbitrarily—*nondeterministic semantics*.

A *nondeterministic mapping* from a domain D_1 to a range D_2 can be represented as a relation $\Pi(D_1, D_2)$. Then we can generalize the notion of monotonicity as follows:

Definition 12. *Monotonicity:* Let $\Pi(D_1, D_2)$ be a nondeterministic mapping, where D_1 and D_2 are domains partially ordered by $\geq_1$ and $\geq_2$, respectively. Then, Π is said to be monotone, when for each $(x, y) \in \Pi$ and every $x' \in D_1$ with $x' \geq_1 x$, there exists a $y' \geq_2 y$ such that $(x', y') \in \Pi$.

Let us now consider transformations defined by our rules. Their domains and range are Herbrand interpretations for the given program; thus, they are both ordered by the set-containment relation $\supseteq$. Then the transformation defined by rule r_2, above, is monotone: if P is a maximal subset of Q satisfying $X \to Y$, and $Q' \supset Q$, then we can construct P' by incrementally adding to P each tuple that does not conflict with X-values of tuples already in P. Then, $(P', Q') \in \Pi$.

Besides the transformation induced by a single rule with choice being monotone, the transformation induced by a positive program with choice is also monotone. This is for instance the case of the transformation induced by the recursive $\mathtt{next}_{\mathtt{r}_1}$ rules, which for each `X` construct a chain that defines a total ordering on the associated `Y`-values in `q(X, Y)`. The properties of positive programs with choice were discussed in [15], where as customary in deductive databases, a program P is viewed as consisting of two separate components: an extensional component, denoted $edb(P)$, and a intensional one, denoted $idb(P)$. Then the following result was proven in [15]:

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 $M' \supseteq M$.*

This theorem allows us to conclude that the recursive rules `next_r1(X, Y, Z)` define a monotone mapping from interpretations of `q(X, Y)` to interpretations of `next_r1(X, Y, Z)`. This mapping is nondeterministic since there are $n!$ orderings for each given set of cardinality n. (The monotonicity of the particular mapping established by the `next_r1(X, Y, Z)` rules can can also be verified directly, since an ordering on a set S can be extended to an ordering on a set $S' \supset S$ by simply attaching the elements of $S' - S$ to the end of the given ordering on S.)

Therefore, a program Q with user-defined aggregates consists of following parts:

1. The `cagr` computation rules
2. The $\mathtt{next}_{\mathtt{r}_1}$ rules defining the nondeterministic monotone ordering transformation
3. The `single`, `multi` and `ereturn`, and `freturn` rules written by the user
4. The early-yield rule,
5. All the remaining rules in the original program Q except r_1,
6. The final-yield rule.

The mapping $\Pi(2^{B_Q}, 2^{B_Q})$ defined by these six sets of rules will be called *the canonical mapping defined by* Q, denoted Π_Q. Assume now that the original program Q has no negated goal and no final-return rule. Then

- Since there is no *final-return* rule, the body of final-yield rule is never made true (enabled) and it can simply be dropped; then
- the remaining rules (i.e., those in points 1-5 above) are positive choice rules, and thus define a nondeterministic monotone mapping.

Therefore:

A UDA that does not have a final-return rule is monotone.

This observation provides a simple and elegant solution to the monotone aggregation problem that has been the subject of significant previous investigations [31,23].

Therefore, the fixpoint-based semantics and computation of positive logic programs can be extended to positive programs with monotone aggregates. We start with a rather obvious definition of the notion of fixpoint for multi-valued mappings:

Definition 13. *Fixpoint:* Let Π denote a mapping and let I be an element in the domain of Π. If $(I, I) \in \Pi$, then I will be called a *fixpoint* for Π.

Definition 14. A *computation* for Π is defined as a sequence

$$I_0, I_1, ..., I_n$$

where, $I_0 = \emptyset$, and for every j, $0 \leq j < n$, $(I_j, I_{j+1}) \in \Pi$ and $I_j \subseteq I_{j+1}$. Here, n can denote a positive integer or the symbol ω.

Definition 15. Let $\langle I_n \rangle$, $n \geq 0$, be a computation for Π. Then the (ω) limit for this computation is defined as follows:

$$lim_{n \geq 0} \langle I_n \rangle \ = \ \bigcup_{n < \omega} I_n$$

Consider now a program Q without negated goals where all the user-defined aggregates have no final-return rule: then Π_Q for this program is monotone. Therefore, *the ω-limit of a computation for such Π_Q is a fixpoint for Π_Q, and vice-versa.* Therefore positive programs with monotone aggregates extend the fixpoint semantics and computation of positive programs.

In conclusion, user-defined aggregates without a final-return rule can be used in positive programs without any restriction, since they have a fixpoint-based semantics that that is based on their stable model semantics. Moreover, bottom-up execution techniques of deductive databases, such as the seminaive fixpoint, remain valid for these programs. Indeed, monotone aggregates can be added to deductive database systems (and were added to $\mathcal{LDL}$++) with minimal effort.

7 Implementation of Extended Aggregates

The implementation of extended aggregates in $\mathcal{LDL}$++ aims at ensuring performance, and upward compatibility with the previous releases (thus, when no return rule is given by the user the system only returns the final result of the aggregate). In the implementation, we use an internal table to store the intermediate results of aggregation. The compilation of aggregate rules is simply a rewriting of these rules into rules without aggregates. Rewritten rules will have two built-in predicates, namely, **readTable** and **writeTable**, which retrieve rows from, and insert rows into, the internal table. The internal table is properly indexed on the group-by columns to speed-up the retrieval and insertion.

An aggregate rule has the following form:

$$\mathtt{p}(\mathtt{K_1}, \cdots, \mathtt{K_m}, \mathtt{aggr_1}\langle\mathtt{A_1}\rangle, \cdots, \mathtt{aggr_n}\langle\mathtt{A_n}\rangle) \leftarrow \mathtt{Rule\ Body}.$$

where $\mathtt{K_1}, \cdots, \mathtt{K_n}$ are group-by columns, $\mathtt{aggr_1}, \cdots, \mathtt{aggr_n}$ are either user defined aggregates or built-in aggregates, and $\mathtt{A_1}, \cdots, \mathtt{A_n}$ are the data the aggregation will be applied to. Both $\mathtt{A_i}$ and $\mathtt{K_i}$ should be bound by goals in the rule body.

The compiler rewrites the above rule into an internal rule as the one shown below; this rule is then linked with the **single**, **multi**, and **return** rules provided by the user. During interpretation, **readTable** is called, and get values from the rule body and from the internal table. The **Flag** argument in **readTable** tells the status of the reading. We have three possible situations:

- Flag=1 (Final State): **readTable** returns a final tuple from the table. Either (1) we have no more tuples from the data source, or (2) the last execution of the **multi** rule fails and either there are no group-by columns in the rule head or the group-by columns are all bound.
- Flag=2 (Initial State): This value is returned from **readTable** when the first tuple encountered with these $\mathtt{K_1}, \cdots, \mathtt{K_m}$ values; in this situation $\mathtt{Aold_1}, \cdots, \mathtt{Aold_n}$ are all `nil`. Then, the **single** rule will be evaluated and the result will be inserted into the table by **writeTable**.
- Flag=3 (Intermediate Recursive States): Here, **readTable** returns the old results which have the same group-by values as the current tuple. Old values are then combined with the current tuple via **multi** rules and the new result is put into the table by **writeTable**.

$$
\begin{array}{l}
\texttt{p}(\texttt{K}_1,\cdots,\texttt{K}_m,\texttt{V}_1,\cdots,\texttt{V}_n) \leftarrow \texttt{readTable}(\texttt{K}_1,\cdots,\texttt{K}_m,\texttt{A}_1,\cdots,\texttt{A}_n,\texttt{Aold}_1,\cdots,\texttt{Aold}_n,\texttt{Flag}),\\
\qquad \texttt{if (Flag} = 1 \texttt{ then}\\
\qquad\qquad \texttt{freturn}(\texttt{aggr}_1,\texttt{A}_1,\texttt{Aold}_1,\texttt{V}_1),\\
\qquad\qquad \ldots\\
\qquad\qquad \texttt{freturn}(\texttt{aggr}_n,\texttt{A}_n,\texttt{Aold}_n,\texttt{V}_n)\\
\qquad \texttt{else}\\
\qquad\qquad \texttt{if (Flag} = 2 \texttt{ then}\\
\qquad\qquad\qquad \texttt{single}(\texttt{aggr}_1,\texttt{A}_1,\texttt{Am}_1),\\
\qquad\qquad\qquad \ldots\\
\qquad\qquad\qquad \texttt{single}(\texttt{aggr}_n,\texttt{A}_n,\texttt{Am}_n),\\
\qquad\qquad \texttt{else}\\
\qquad\qquad\qquad \texttt{multi}(\texttt{aggr}_1,\texttt{A}_1,\texttt{Aold}_1,\texttt{Am}_1),\\
\qquad\qquad\qquad \ldots\\
\qquad\qquad\qquad \texttt{multi}(\texttt{aggr}_n,\texttt{A}_n,\texttt{Aold}_n,\texttt{Am}_n)),\\
\qquad\qquad \texttt{writeTable}(\texttt{K}_1,\cdots,\texttt{K}_m,\texttt{Am}_1,\cdots,\texttt{Am}_n),\\
\qquad\qquad \texttt{ereturn}(\texttt{aggr}_1,\texttt{A}_1,\texttt{Aold}_1,\texttt{V}_1),\\
\qquad\qquad \ldots\\
\qquad\qquad \texttt{ereturn}(\texttt{aggr}_n,\texttt{A}_n,\texttt{Aold}_n,\texttt{V}_n)).
\end{array}
$$

8 Applications of Monotone Aggregation

Consider the following rule that uses the aggregate `mcount`, which returns the current count after each new value;

$$\texttt{q}(\texttt{mcount}\langle\texttt{X}\rangle) \leftarrow \texttt{p}(\texttt{X}).$$

If `p` is a database predicate with $n \geq 1$ facts, then the rule returns $I_n = \{q(1), q(2), \ldots, q(n)\}$. If the original set of facts is increased to a new set of cardinality $m \geq n$, then, our rule returns: $I_m = \{q(1), q(2), \ldots, q(m)\}$, where $I_m \supseteq I_n$. *Therefore rules with* `mcount` *define a monotone deterministic mapping.*

All aggregates inductively defined using single, multi and early-return rules define monotone mappings. In general, these mapping are nondeterministic (unlike `mcount`). The examples which follow are based on those in [23], and show that, deterministic or otherwise, monotone aggregates provide a powerful and flexible tool for advanced applications.

Join the Party Some people will come to the party no matter what, and their names are stored in a `sure(Person)` relation. But others will join only after they know that at least K of their friends will be there. Here, `friend(A,B)` denotes that B views A as a friend.

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

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).
sure(tom).
sure(jane).

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

Then, the basic seminaive computation yields:

```
willcome(mark).
willcome(tom).
willcome(jane).

c_friends(jerry,1).
c_friends(penny,1).
c_friends(jerry,2).
c_friends(penny,2).
c_friends(penny,3).

willcome(penny).

c_friends(jerry,3).

willcome(jerry).
```

This example illustrates how the standard seminaive computation [35] can be applied in conjunction with monotone user-defined aggregates.

The use of the aggregate `atleast` simplifies the computation of the previous query significantly. Our program now becomes:

```
wllcm(F,yes) ←                  sure(F).
wllcm(X,atleast⟨(3,F)⟩) ← wllcm(F,yes),friend(X,F).
```

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.

Next, we define `msum` and `mmin` that provide monotone extensions for `sum` and `min`. A `mmax` aggregate can be defined in a similar way. All these aggregates are monotonic and nondeterministic.

For msum we have:

$$\texttt{single(msum, Y, Y).}$$
$$\texttt{multi(msum, Y, Old, New)} \leftarrow \texttt{New} = \texttt{Old} + \texttt{Y}.$$
$$\texttt{ereturn(msum, Y, Old, Z)} \leftarrow \texttt{if(Old} \neq \texttt{nil then Z} = \texttt{Old} + \texttt{Y else Z} = \texttt{Y)}.$$

For mmin we have:

$$\texttt{single(mmin, Y, Y).}$$
$$\texttt{multi(mmin, Y, Old, New)} \leftarrow \texttt{if(Y} < \texttt{Old then New} = \texttt{Y else New} = \texttt{Old)}.$$
$$\texttt{ereturn(mmin, Y, Old, Y)} \leftarrow \texttt{if(Old} \neq \texttt{nil then Y} < \texttt{Old)}.$$

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

Least-Distance Connections For instance, given a graph g(X, Y, C) where C is the cost of an edge from node X to node Y, the least-cost path between any two nodes can be computed as follows:

$$\texttt{ld(X, Y, mmin}\langle\texttt{C}\rangle\texttt{)} \leftarrow \quad \texttt{g(X, Y, C).}$$
$$\texttt{ld(X, Y, mmin}\langle\texttt{C}\rangle\texttt{)} \leftarrow \quad \texttt{ld(X, Z, C1), ld(Z, Y, C2), C} = \texttt{C1} + \texttt{C2}.$$
$$\texttt{least\_dist(X, Y, min}\langle\texttt{C}\rangle\texttt{)} \leftarrow \texttt{ld(X, Z, C1).}$$

This transitive-closure like computation adds a new arc ld(X, Y, C) provided that this then becomes the new least-cost arc between the nodes X and Y. The arcs so produced are then used in the next step of the seminaive computation. At the end of this fixpoint computation, the least_dist rule is used to select the least-cost arc between these two nodes, out of the succession of arcs of decreasing C values produced in the computation.

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 in least_dist, however, are always the same (a nondeterministic computation producing a deterministic answer). In fact, the presence of the least_dist rule suggests that we might replace mmin⟨C⟩ by simply C without changing the final result. However, in the situation where there are cycles of positive cost, our program terminates whereas the other will not. Also in a DAG, where longer paths correspond to higher costs, then our program performs significantly better than the other program.

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, 49% 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:

$$
\begin{array}{ll}
\texttt{control(C,C)} \leftarrow & \texttt{owns(C,\_,\_)}. \\
\texttt{control(Onr,C)} \leftarrow & \texttt{twons(Onr,C,Per),Per} > 49. \\
\texttt{towns(Onr,C2,msum}\langle\texttt{Per}\rangle) \leftarrow & \texttt{contrl(Onr,C1),owns(C1,C2,Per)}.
\end{array}
$$

Thus, every company controls itself, and a company `C1` that has transitive ownership of more than 49% of `C2`'s shares controls `C2` . In the last rule, `twons` 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.

To further simplify the program and expedite the computation we can introduce a special aggregate as follows [1]:

$$
\begin{array}{ll}
\texttt{single(sum49,Y,Y)}. & \\
\texttt{multi(sum49,Y,Old,Z)} \leftarrow & \texttt{Old} < 49, \texttt{Z} = \texttt{Old} + \texttt{Y}. \\
\texttt{ereturn(sum49,Y,Old,yes)} \leftarrow & \texttt{if(Old} = \texttt{nil then Y} > 49 \\
 & \quad \texttt{else Old} + \texttt{Y} > 49).
\end{array}
$$

Then the recursive rules become:

$$
\begin{array}{ll}
\texttt{cntrl(Onr,C,yes)} \leftarrow & \texttt{owns(Onr,C,Per),Per} > 49. \\
\texttt{cntrl(Onr,C2,sum49}\langle\texttt{Per}\rangle) \leftarrow & \texttt{cntrl(Onr,C1),owns(C1,C2,Per)}.
\end{array}
$$

Thus, the `sum49` aggregate succeeds only when the 49% threshold is crossed during the summation. Here, the value of 49 was cast into the very definition of our aggregate. Alternatively, this value could be given as a parameter, as in the case of `atleast`.

Bill-of-Materials (BoM) Applications BoM applications represent an important application area that requires aggregates in recursive rules. Say, for instance that `psb(P1,P2,QT)` denotes that `P1` contains part `P2` in quantity `QT`. We also have elementary parts that are purchasable for a price and will be delivered in a certain number of days: these are described by the relation `basic(P,Price,Days)`. Then, the following program computes the cost of a part as the sum of the costs of the basic parts it contains.

$$
\begin{array}{l}
\texttt{part\_cost(Part,0,Cst)} \leftarrow \quad \texttt{basic(Part,Cst)}. \\
\texttt{part\_cost(Part,mcount}\langle\texttt{Sb}\rangle\texttt{,msum}\langle\texttt{MCst}\rangle) \leftarrow \\
\qquad\qquad \texttt{part\_cost(Sb,ChC,Cst),prolfc(Sb,ChC)}, \\
\qquad\qquad \texttt{psb(Part,Sb,Mult),MCst} = \texttt{Cst} * \texttt{Mult}.
\end{array}
$$

Thus, the key condition in the body of the second rule is that a subpart `Sb` is counted in `part_cost` only when all `Sb`'s children have been counted.

[1] The aggregate `sum49` is deterministic

This occurs when the number of `Sb`'s children counted so far by `mcount` is equal to its total number of children in the `psb` graph. This last number is kept in the prolificity table, `prolfc`, which can be computed as follows:

$$\begin{array}{ll} \texttt{prolfc(P1,count}\langle\texttt{P2}\rangle\texttt{)} \leftarrow & \texttt{psb(P1,P2,\_).} \\ \texttt{prolfc(P1,0)} \leftarrow & \texttt{basic(P1,\_).} \end{array}$$

Also this BOM computation can be simplified and made more efficient using the `atleast` aggregate, yielding:

$$\begin{array}{l} \texttt{pcost(Part,yes,Cst)} \leftarrow \quad \texttt{basic(Part,MCst).} \\ \texttt{pcost(Part,atleast}\langle\texttt{(K,Sb)}\rangle\texttt{,msum}\langle\texttt{Cst}\rangle\texttt{)} \leftarrow \\ \qquad\qquad \texttt{pcost(Sb,yes,Cst),psb(Part,Sb,Mult),} \\ \qquad\qquad \texttt{prolfc(Part,K),MCst} = \texttt{Cst} * \texttt{Mult.} \end{array}$$

An improvement with respect to the previous formulation is that the `prolfc` relation is now used to qualify `Part` as they are counted in the the rule head.

The technique of counting the children could also be used with `least_dist` problem, above, 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 (idempotence).

9 Applications to SQL Databases

Encouraged by our success with $\mathcal{LDL}$++, we have designed and implemented a system called SQL-AG that supports SQL3 user-defined aggregates and actually extends it by supporting early returns. Then we explored the use of these new aggregates to express complex data mining algorithms, with encouraging results: complex algorithms, such as iceberg queries [11], and the SPRINT [2] algorithm enhanced with the PUBLIC [28] technique recently proposed for pruning, were quickly coded using special user defined aggregates

We have actually developed two versions of SQL-AG. The first is for Oracle DBMSs using P/SQL and is described in [4]. Here we describe the second system built on DB2 using user-defined scratch-pad functions. SQL-AG implements SQL3 specification for user defined aggregates on DB2. For example, a user can define an iterative routine for average as follows:

```
AGGREGATE FUNCTION myavg( IN NUMBER)
RETURNS NUMBER
STATE NumberPair
INITIALIZE average-init
ITERATE average-iterate
TERMINATE average-terminate
```

Here average-init, average-iterate and average-terminate are functions that the user will have to define in C. When the definition of an aggregate, such as myavg, is given, SQL-AG generates a C source file which defines the two user-defined functions, myavg() and myavg_groupby(), to implement the new aggregate. These two functions are then compiled and linked with DB2's library to support the new aggregate.

Once registered, user-defined aggregates can be used in SQL statements as if they are standard built-in aggregates (*sum, min, max, avg, count*). Take for instance the following SQL-AG query:

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

The SQL-AG system transforms the above query into the following equivalent query that is then execute on DB2:

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

Thus, the function myavg_groupby is applied on very record for aggregation. This calls avg_multi and then updates the current values of state_type for dept that are kept in an hash table. Then myavg_groupby returns the value zero. DB2 applies the actual GROUP BY clause at the end of the scan. Thus, myavg(dept) in the select clause returns the results left in the hash table by the previous computation.

SQL-AG is quite general and efficient. Indeed, we have evaluated the efficiency of our implementation by coding builtins such as sum or avg in SQL-AG comparing performance. We tested performance of builtins computed in three different ways: (i) the original SQL builtin, (ii) the UDA running in fenced mode, and (iii) the UDA running in unfenced mode. User-defined aggregates running in fenced mode are almost 10 times slower than builtin aggregates. Unfenced UDAs are much better, since their performance is very close (10 percent above) that of the builtins. Thus the flexibility of supporting early returns is not bought at the price of lesser performance.

Aggregates with Early Returns Online aggregation [17] exemplifies the new kind of column functions required by advanced applications, particularly data mining ones. For instance, the computation of averages often converges toward the final value after a small subset of input data is visited. Thus, it is desirable to produce "early returns" during the computation. Therefore, a new function called PRODUCE has been introduced in SQL-AG that will return values during the computation. For instance an online version of myavg can be defined as follows:

```
AGGREGATE FUNCTION online_avg( IN NUMBER)
RETURNS return_type
STATE number-pair
INITIALIZE online_avg_single
ITERATE online_avg_multi
PRODUCE online_avg_return
TERMINATE NOP
```

Here, the functions for INITIALIZE, ITERATE, are basically the same as those in myavg, and the newly introduced PRODUCE calls an online_avg_return function defined in C that returns a partial result every five new input values. This example illustrates a situation of particular significance, where is the situation where only early returns are used (i.e., TERMINATE is declared NOP or is missing) since *the aggregate function so defined is monotonic.* Thus, SQL3's limitation that queries must be stratified with respect to aggregates no longer applies. This allows us to support efficiently a host of applications that have been a problem for SQL. The following example [23] uses a monotonic count aggregate mcount defined in C, but behaving exactly as the `mcount` aggregate previously discussed for $\mathcal{LDL}$++.

Join the Party Some people will come to the party no matter what, and their names are stored in a sure(Pname) relation. But many other persons will join only after they know that at least 3 of their friends will be there. Here, friend(Pname, Fname) is a table denoting that Pname views Fname as a friend.

Example 1. Join the Party in SQL-AG

```
WITH 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
```

SQL-AG implementation in DB2 of this query is shown below. Observe that we basically use the same translation as that of final returns in TERMINATE, modulo the absence of the GROUP BY clause. Thus, early returns from PRODUCE are returned immediately in the SELECT clause, rather than after the GROUP BY operation used for final returns.

Example 2. The DB2 implementation for the Join-the-Party Query

```
WITH wllcm(Name) AS
(   SELECT Pname FROM sure
UNION ALL
    SELECT f.PName
    FROM wllcm, friend f
    WHERE wllcm.Name = f.Fname
    AND mcount_groupby(f.Pname,f.Fname)=0
    AND mcount(f.Fname)=3
)
SELECT Name
FROM wllcm
```

10 Conclusions

In this paper, we have provided the logic-based foundations for the extended aggregate capability required by the next generation of database applications. In particular, we focused on data mining applications and showed that user-defined aggregates extended with early returns are critical to support a wide variety of applications ranging from online estimates to iceberg queries. Then, we proposed a simple formalization of our extended aggregates using programs with the nondeterministic construct of choice—a class of programs that combine total stable model semantics with efficient operational semantics. Furthermore, this formalization leads to a simple syntactic characterization of monotone user-defined aggregates. Since monotone aggregates can be used freely in recursive rules, complex queries and applications can now be expressed with simple programs amenable to efficient implementation.

Our results underscore the significance of logic in databases—a significance that was paramount in the original formulation of relational databases by Codd, but was much less so in more recent developments. In fact, in the thirty years that have passed since the introduction of the relational model, critics have often complained about the inability of relational databases to express critical queries, such as those required in BoM applications. The great work on recursion by deductive database researchers lead to the introduction of recursive queries in SQL3, but has not helped much with those queries. Indeed, most BoM queries require the computation of aggregates during the recursive traversal of the part/subpart graph; but, in SQL3, all queries must be stratified with respect to recursion and aggregates, and the same holds for most deductive database systems. Therefore, the aggregates-in-recursion challenge has been the focus of some of the best research work in the deductive database area [8,7,22,20,31,34,23]. Of particular interest, there is the work by Ross and Sagiv [23] who showed that, for each aggregate, a particular lattice can often be found, where this aggregate defines monotone transformations. The difficulty of automatically identifying such lattices [31] and deriving lattice-specific fixpoint computations prevented their implementation in actual systems. Given this background, our discovery of monotone aggregates that preserve the lattice and fixpoint computation techniques of positive programs represents an unexpected breakthrough.

References

1. Abiteboul S., Hull R. and Vianu V., *Foundations of Databases*, Addison Wesley, 1995.
2. R. Agrawal and R. Srikant, "Fast Algorithm for Mining Association Rules." In *Proceedings of the 20th VLDB Conference*, Santiago, Chile, 1994.
3. R. Agrawal and J.C. Shafer, "Parallel Mining of Association Rules: Design, Implementation and Experience", *IEEE Transactions on Knowledge and Data Engineering*, Vol.8, No. 6, Dec. 1996.
4. P. Barghava, "User-Defined Aggregates in Database Languages," MS Thesis, UCLA, 1998: `http://www.cs.ucla.edu/czdemo/SQL-AG/`.
5. S. Brin, Rl Motwani, J. Ullman, and S. Tsur, "Dynamic Itemset: Counting and Implication Rules for Market Basket Data, *Proceedings ACM-SIGMOD Int. Conf on Management of Data*, May 1997.
6. C. X. Chen and C. Zaniolo, "Universal Temporal Data Languages," DDLP'98 Workshop, 1998.
7. M. P. Consens and A. O. Mendelzon, "Low Complexity Aggregation in Graphlog and Datalog," ICDT'90, 1990.
8. I. F. Cruz and T. S. Norvell, "Aggregative Closure: an Extension of Transitive Closure," *Proc.Fifth Int. Conference on Data Engineering*, pp. 384-389, 1989.
9. D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur, C. Zaniolo: The LDL System Prototype. *IEEE Transactions on Knowledge and Data Engineering* 2(1): 76-90 (1990)
10. Charles Elkan. "Boosting and Naive Bayesian Learning." Technical report no cs97-557, Dept. of Computer Science and Engineering, UCSD, September 1997.
11. M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motawni, J. Ullman, "Computing Iceberg Queries Efficiently", *Proceedings of the 1998 VLDB Conference*, New York, NY, 1998.
12. J. Gray, A. Bosworth, H. Pirahesh, A. Layman. "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total." *Proc. International Conference on Data Engineering*, 1996.
13. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. newblock *Proceedings of Joint International Conference and Symposium o n Logic Programming*, pp. 1070–1080, Seattle, WA, 1988.
14. Gyssen, M., Van Gucht, D. and Badia, A., Query Languages with Generalized Quantifiers, in *Applications of Logic Databases*, R. Ramakrishan, Kluwer, 1995.
15. F. Gianotti, D. Pedreschi, and C. Zaniolo, "Semantics and Expressive Power of Non-Deterministic Constructs in Deductive Databases," *JCSS* to appear.
16. S. Greco and C. Zaniolo, "Greedy Algorithms in Datalog with Choice and Negation" *JICSLP'98*, Manchaster UK, June 16-19, 1998.

17. J. M. Hellerstein, P. J. Haas, and H. J. Wang. "Online Aggregation." In *Proceedings of ACM-SIGMOD Conference on Management of Data*, May 1997.
18. Hsu, P. Y. and Parker, D. S., Improving SQL with Generalized Quantifiers, *Proc. ICDE*, 1995.
19. "Database Language SQL - Part 2: SQL/Foundation.", July 1996.
20. David B. Kemp and Peter J. Stuckey. "Semantics of logic programs with aggregates" *Proceedings of the 1991 International Symposium on Logic Programming*, pages 387–401, October 1991.
21. I. Motakis and C. Zaniolo, "Temporal Aggregation in Active Database Rules." *ACM SIGMOD*, 1997.
22. I. S. Mumick, H. Pirahesh, and R. Ramakrishan, "The magic of duplicates and aggregates," *VLDB 1990*, pp. 264-277, 1990.
23. K. A. Ross and Yehoshua Sagiv, "Monotonic Aggregation in Deductive Database", *JCSS*, 54(1), 79-97 (1997)
24. L. Rowe and M. Stonebraker. "The POSTGRES Data Model." In *Proceedings of VLDB Conference*, Brighton, England, September 1987.
25. 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.
26. D. Saccà and C. Zaniolo. Deterministic and non-deterministic stable models. Journal of Logic and Computation, 7(5):555-579, October 1997.
27. S. Sarawagi, S. Thomas, R. Agrawal, "Integrating Association Rule Mining with Relational Database Systems: Alternatives and Implications," *Proceedings ACM-SIGMOD Int. Conf on Management of Data*, pp.343-354, May 1999.
28. K. Shim and R. Rastogi, "PUBLIC: A Decision Tree Classifier that Integrates Building and Pruning" *VLDB'98*, New York City, N.Y., 1998
29. W. Shen, K.Ong, B. Mitbander and C. Zaniolo, Metaqueries for Data Mining, Chapter 15 of *Advances in Knowledge Discovery and Data Mining*, U. M. Fayyad et al (eds.), MIT Press, 1996.
30. M. Stonebraker, L. Rowe, and M. Hirohama. "The Implementation of POSTGRES." *IEEE Transactions on Knowledge and Data Engineering*, **2**(1), March 1990.
31. A. Van Gelder. Foundations of Aggregations in Deductive Databases In *Proc. of the Int. Conf. On Deductive and Object-Oriented databases*, 1993.
32. Keenan, E. and Westertahl, D., Generalized Quantifiers in Linguistic and Logic, In *Generalized quantifiers in Natural Language*, van Benthem and ter Meulen (eds.) Foris Publications, 1985.
33. Carlo Zaniolo. "Design and Implementation of a Logic Based Language for Data Intensive Applications." In *Proceeding of the International Conference on Logic Programming*, 1988.
34. Carlo Zaniolo, N. Arni, and K. Ong. "Negation and Aggregates in Recursive Rules: the LDL++ Approach." *DOOD*, 1993.
35. C. Zaniolo, S. Ceri et al., *Advanced Database Systems*, Morgan Kaufmann Publishers, 1997.