Aggregates in Recursive Datalog and SQL3 Queries

Haixun Wang and **Carlo Zaniolo**

University of California at Los Angeles
Los Angeles, CA 90095
hxwang—zaniolo@cs.ucla.edu

Abstract

Advanced database applications require user-defined aggregates that, along with the final returns provided by traditional aggregates, also support *early returns* as needed in online aggregation and many data mining applications.

In this paper, we (i) define the formal semantics of user-defined aggregates in Datalog, (ii) introduce aggregates with early returns in $\mathcal{LDL}$++ and SQL3, and (iii) provide a simple syntactic characterization of aggregates that are monotonic under set-containment. Thus, we find that aggregates functions based on early returns are monotone and, therefore, can be freely used in recursive queries. This allow us to succinctly express and efficiently compute queries, such as Bill-of-Materials or least-distance graph traversals, that have traditionally been recalcitrant to relational query languages—and to SQL3's recursive queries as well, since these are restricted to a stratified usage of traditional nonmonotonic aggregates. Finally, we describe our implementations of these aggregates in $\mathcal{LDL}$++, and in object/relational systems—including an efficient implementation using DB2's scrath-pad functions.

1 Overview

A new wave of database applications, particularly data mining applications, require aggregates much more powerful than the five builtins supported in SQL2. Several vendors have moved to provide additional aggregates, such as Data Cubes [9], but the range and variety of functions required by new applications can only be accommodated by user-defined aggregates (UDAs): Postgres [18] and SQL3 [12] already support UDAs. For instance, SQL3 prescribes that a new aggregate should be defined via the following three iterative routines, that are to be coded in and imported from a procedural language:

- an INITIALIZE routine to generate the initial value for the computation of the UDA,
- an ITERATE routine yielding a new value for each new record in the table, and
- a TERMINATE routine that returns the final value for the aggregate.

The new-generation Object/Relational databases, such as DB2, support user defined scalar functions, but not user-defined column functions (as aggregates are called in DB2). Furthermore, the current SQL3 proposal does not support the important notion of online aggregation [10]. Indeed, this very useful extension requires that partial results and *early returns* be produced *during* the computation, rather than at the end of the computation as in SQL3 TERMINATE.

A substantial amount of previous work has focused on aggregates and their semantics in logic-based programs [3, 2, 14, 13, 15, 19]. Of particular significance is the work on monotonic aggregation [15], since monotonic aggregates can be used to express very useful recursive queries (e.g., for Bill of Materials and least-distance paths in a graph) that have posed a long-standing challenge to relational DB languages. As shown in [15, 19], many aggregates are monotonic under partial orders that are different from the set-containment ordering used to derive the fixpoint for the standard T_P operator. This, and the fact that automatic derivation of partial orders for arbitrary aggregates is very difficult [19], prevented their use in Datalog systems and SQL: SQL3 prescribes that aggregates can only be used in stratified queries [4, 12].

This paper presents a condensed solution to all the before-mentioned problems, since we (i) define the formal semantics of user-defined aggregates in Datalog languages (ii) extend $\mathcal{LDL}$++ and SQL3 with the constructs needed to express early returns, (iii) provide a simple syntactic characterization of aggregates that are monotonic under set-containment, and (iv) describe our efficient implementations of these extended UDAs in $\mathcal{LDL}$++ and DB2.

2 Logic-Based Languages

In CORAL and $\mathcal{LDL}$++, the following rule can be used to add up the salaries of employees grouped by department:

$$\texttt{addSalbyDept(DeptNo, sum}\langle\texttt{Sal}\rangle\texttt{)} \leftarrow \quad \texttt{empl(Ename, Sal, DeptNo)}.$$

Thus, aggregates, such as $\texttt{sum}\langle\ldots\rangle$, are used as arguments in the head of the rule, and the remaining non-aggregate arguments are interpreted as group-by attributes. In addition to SQL's builtins, $\mathcal{LDL}$++, supports the rule-based definition of UDAs. In general, a new aggregate function F can be defined inductively by specifying:

1. [*Base*] A rule for computing the value F on a singleton set $\{x\}$, and
2. [*Induction*] A rule for computing F on a set $S' = S \sqcup \{x\}$ as a function of $F(S)$ and x.

In $\mathcal{LDL}$++, the rules defining the computation of the base and induction steps are respectively named `single` and `multi`. Thus, `count` and `sum` can be defined by following rules:

For `count`:

$$\texttt{single(count, Y, 1).}$$
$$\texttt{multi(count, Y, Old, New)} \leftarrow \quad \texttt{New} = \texttt{Old} + \texttt{1}.$$

For `sum`:

$$\texttt{single(sum, Y, Y).}$$
$$\texttt{multi(sum, Y, Old, New)} \leftarrow \quad \texttt{New} = \texttt{Old} + \texttt{Y}.$$

The original version of $\mathcal{LDL}$++ developed at MCC only returns results at the end of the computation (same as SQL3), and therefore does not support online aggregation and other aggregates requiring early returns. To cure this problem, at UCLA we have developed a new version of $\mathcal{LDL}$++ that, along with `single` and `multi` previously discussed, supports the predicates `ereturn` and `freturn` to specify values produced as early returns and as final returns, respectively. For instance, the following aggregate `onl_avg` returns the current estimate for average every five elements.

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

If the final value of the average is also desired, the following rule can be added:

$$\texttt{freturn(onl\_avg, X, (Sum, Count), Avg)} \leftarrow \texttt{Avg = Sum/Count.}$$

The early-return capability was initially introduced to support on-line aggregation, but then we found that it makes it easy to introduce many new aggregates which would be impossible (or prohibitively difficult and inefficient) to express without it. Indeed, we implemented an assortment of complex aggregates including: iceberg queries, median estimation, roll-ups, temporal aggregates for time series, set-comparison aggregates, and Allen temporal operators; for a detailed discussion, and live demo see `http://www.cs.ucla.edu/ldl`.

For instance, let us define an extreme version of online count, i.e., one that returns the incremental count at each step. The new aggregate called `mcount` is defined as follows:

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

The first `ereturn` rule applies when `Old = nil`; this condition 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, 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:

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

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

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

Then, a predicate equivalent to the $\texttt{large\_dept(D\#)}$ can be formulated as follows:

$$\texttt{lrg\_dpt(D\#, atleast}\langle(\texttt{7, Ename})\rangle\texttt{)} \leftarrow \quad \texttt{empl(Ename, Sal, D\#)}.$$

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 $?\texttt{lrg\_dpt(D\#, yes)}$, for which no existential optimization is performed.

Observe that `mcount` and `atleast` aggregates are monotonic: as the relations in the body of the previous rules defining `count_emp` and `lrg_dpt` become larger, so do the results in the heads of the beforementioned rules. In fact, the formal semantics of aggregates is such that *online aggregates are monotonic, and can be freely used in recursive rules.*

Before we formalize the semantics of aggregates, consider a moving-window aggregate for time-series analysis. Say that we have sequences of closing prices for stocks, and we want to average the prices of IBM over the last five days.

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

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

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

3 Formal Semantics and Monotonicity

The formal semantics of a program with aggregates can be defined by viewing it as a short-hand for an expanded program without aggregates. For instance, say that we have the following rule where we compute a new aggregate function `myagr` on a set of `Y`-values grouped by `X`:

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

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 require nonomonotonic constructs to visit the elements on-by-one, thus preventing the use of aggregates in recursive rules. A better solution consists in using `choice` [16, 17] 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 [6].

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

Ordering Rules:

$$\begin{array}{ll} \texttt{next\_r}_1\texttt{(X,nil,nil)} \leftarrow & \texttt{q(X,Y)}. \\ \texttt{next\_r}_1\texttt{(X,Y1,Y2)} \leftarrow & \texttt{next\_r}_1\texttt{(X,\_,Y1),q(X,Y2)}, \\ & \texttt{choice((X,Y1),(Y2)),choice((X,Y2),(Y1))}. \end{array}$$

Aggregates can be defined by the following internal recursive predicate `cagr`:

<u>`cagr Rules`</u>

$$\begin{array}{ll} \texttt{cagr(myagr,X,Y,New)} \leftarrow & \texttt{next\_r}_1\texttt{(X,nil,Y)}, \texttt{Y} \neq \texttt{nil}, \texttt{single(myagr,Y,New)}. \\ \texttt{cagr(myagr,X,Y2,New)} \leftarrow & \texttt{next\_r}_1\texttt{(X,Y1,Y2),cagr(myagr,X,Y1,Old)}, \\ & \texttt{multi(myagr,Y2,Old,New)}. \end{array}$$

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):

<u>Single, Multi, Early-Returns, and Final-Returns Rules</u>

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

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:

<u>Early-Yield Rule:</u>

$$\begin{array}{ll} \texttt{p(X,AgrVal)} \leftarrow & \texttt{next\_r}_1\texttt{(X,nil,Y)}, \texttt{Y} \neq \texttt{nil}, \texttt{ereturn(myagr,Y,nil,AgrVal)}. \\ \texttt{p(X,AgrVal)} \leftarrow & \texttt{next\_r}_1\texttt{(X,Y1,Y2),cagr(myagr,X,Y1,Old)}, \\ & \texttt{ereturn(myagr,Y2,Old,AgrVal)}. \end{array}$$

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

<u>Final-Yield Rule:</u>

$$\begin{array}{ll} \texttt{p(X,AgrVal)} \leftarrow & \texttt{next\_r}_1\texttt{(X,\_,Y)}, \neg\texttt{next\_r}_1\texttt{(X,Y,\_)}, \texttt{cagr(myagr,X,Y,Old)}, \\ & \texttt{freturn(myagr,Y,Old,AgrVal)}. \end{array}$$

Therefore, the meaning of a program P containing a rule such as $\texttt{r}_1$, is simply the program P' obtained by replacing $\texttt{r}_1$ 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* [7].

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 multivalued mapping from $edb(P)$ to $idb(P)$, as per the following theorem proven in [7]:

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 multivalued 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 multivalued mapping also induces a nondeterministic (single-valued) mapping, defined as an arbitrary choice among the values of the function. As shown in [7], 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 [6] of the program obtained by rewriting the choice program into an equivalent program with negation [16]. Therefore, keeping with previous authors [13], we have defined the semantics of aggregates in terms of stable models [6]; however, through the use of the choice construct, we have avoided the computational intractability problems of stable models. Indeed in our formal semantics of aggregates, choice rules are only used to deliver the `next` value `Y2` generated by our rule $\mathtt{r}_1$(for the given group-by value `X` and the previous such value `Y1`): thus, an operational realization is very simple and efficient.

In summary, we have that (i) user-defined aggregates without final-returns can be used in recursive programs without any restriction, since their equivalent choice programs define programs that always have total stable models, and (ii) such stable models can be derived using an inflationary fixpoint computation [7]. Finally, since aggregates define monotonic transformations in the usual lattice of set containment, bottom-up execution techniques of deductive databases, such as the seminaive fixpoint, remain valid for these programs. In conclusion, monotone aggregates can be added to deductive database systems with no change in their execution strategy; indeed, their addition to $\mathcal{LDL}$++ only required a modest effort. As we shall see later, similar encouraging results hold for recursive SQL3 queries. (The semantic and monotonicity characterization presented here hold independent of the constructs used in coding `single`, `multi`, `ereturn`, and `freturn`; thus we can use lists, as in `mw5avg`, or other Turing completeness inducing constructs. Therefore, the use of procedural routines, such as those used in SQL3 UDAs, will not compromise the monotonicity of such UDAs.)

4 Programs with Monotone Aggregation

All examples given in [15] were implemented in the new version of $\mathcal{LDL}$++ developed at UCLA. Some simple cases are discussed next.

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(P, F)` denotes that the person `P` views the person `F` as a friend.

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

Here, $K = 3$ is the number of friends required by any person to come to the party. This example can be further simplified using `atleast`:

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

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.

While `mcount` and `atleast` define monotonic deterministic mapping, other aggregates such as `mmin`, below, are monotonic but nondeterministic.

$$\begin{array}{ll} \texttt{single(mmin,Y,Y)}. & \\ \texttt{multi(mmin,Y,Old,Y)} \leftarrow & \texttt{Y} < \texttt{Old}. \\ \texttt{multi(mmin,Y,Old,Old)} \leftarrow & \texttt{Y} \geq \texttt{Old}. \\ \texttt{ereturn(mmin,Y,Old,Y)} \leftarrow & \texttt{Y} \neq \texttt{Old}. \end{array}$$

Thus a new value `Y` is returned whenever a new partial minimum is found.

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 by transitive-closure like computation that 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.

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

At the end of the fixpoint computation, the usual aggregate `min` was used to select the least-cost arc between each pair of nodes, out of the succession of arcs of decreasing `C` values produced during the computation. Thus, for an acyclic graph the final result will not change if we simply replace '$\texttt{mmin}\langle\texttt{C}\rangle$' by '`C`'. But if there are cycles of positive cost, our program terminates whereas the other does not. Also in a DAG, where longer paths correspond to higher costs, our program performs significantly better.

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.

5 SQL-AG

We developed two prototype implementations for SQL3 UDAs. One uses PL/SQL on Oracle [5]. Here we describe the second implementation that builts on DB2, using the scratch-pad user defined functions (UDFs) supported by this object/relational system.

To define a new aggregate in SQL-AG, say `myavg`, a user has to write three functions in a language such as C. Say that such functions were defined and called avg_single, avg_multi, and avg_result. Now the user must register the new aggregate using the following SQL3-compliant declarations:

```
AGGREGATE FUNCTION myavg( IN NUMBER)
RETURNS return_type
STATE state_type
INITIALIZE avg_single
ITERATE avg_multi
TERMINATE avg_result
```

SQL-AG parses this declaration and generates the source C file for two UDFs, mamed myavg() and myavg_groupby(). Then, SQL-AG compiles and links these two UDFs with DB2's library. After this step, the UDA `myavg` can be used in SQL/DB2 statements as any builtin aggregate. For instance the user could write the following query:

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

SQL-AG transforms this query into the following equivalent query excutable on DB2:

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

Thus, the UDF myavg_groupby is applied on every record for aggregation. This calls avg_multi and then updates the current values of state_type for dept that are kept in a 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's support for SQL3 UDAs is quite general and efficient. In Example 1 we show how a query involving two user-defined aggregates and having conditions is supported by SQL-AG (computing the sum of the two functions ensures that they are both called):

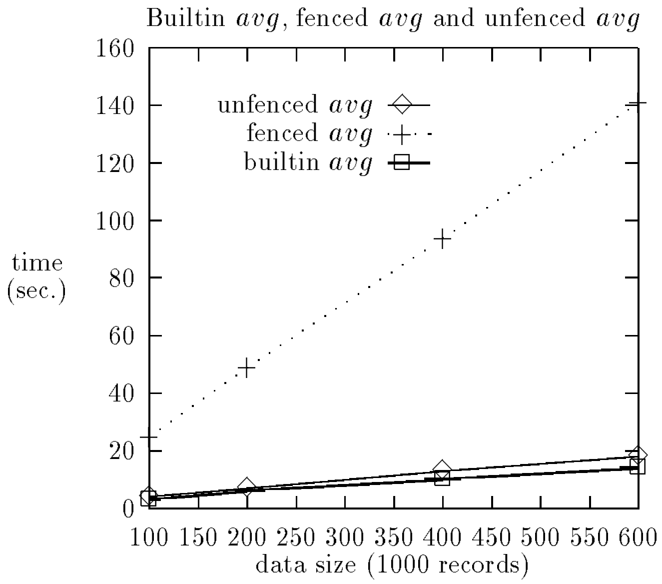
Example 1 *Transforming SQL Statement with Having Clause*

```
SELECT dept, mycnt(name)
FROM emp
WHERE dept <> 'Accounting'
GROUP BY dept
HAVING myavg(salary)>100
```

```
SELECT dept, mycnt(dept)
FROM emp
WHERE dept <> 'Accounting'
   AND mycnt_groupby(dept,name) +
         myavg_groupby(dept,salary) = 0
GROUP BY dept
HAVING myavg(dept)>100
```

We compared the performance of native DB2 builtins against SQL-AG UDAs.

The performance curves compare (i) the SQL builtin `avg`, (ii) the user-defined `myavg` running in fenced mode, and (iii) the user-defined `myavg` running in unfenced mode. UDAs running in fenced mode are almost 10 times slower than builtin aggregates. However, the performance of the UDA myavg(X) running in unfenced mode is very close (within 10%) to the builtin avg(X). Similar results hold for the other builtins. Thus the flexibility of supporting UDAs in SQL-AG is not bought at the price of lesser performance.



6 Early Returns

SQL-AG also supports early returns, by allowing the user to add an external PRODUCE function to the INITIALIZE, ITERATE, and TERMINATE functions of SQL3. For instance, an on-line version of `myavg` would also contain the following line:

PRODUCE online_avg_return

Here, online_avg_return is a C function which produces early returns, according to some user-coded criterion. The definition for myavg would still contain a TERMINATE function, if we also want the final result returned. If the final result is of no interest, then TERMINATE line can be omitted, or the user can simply write TERMINATE NOP. When TERMINATE is either missing or NOP, SQL-AG recognizes this as a *monotonic aggregate* and *supports its unrestricted use in recursive SQL3 queries*. The following example uses an aggregate mcount, with the same meaning of that used in $\mathcal{LDL}$++ but encoded in C.

Example 2 *Join the Party in SQL-AG and its implementation in DB2*

```
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
```

```
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
```

Observe that for PRODUCE 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. The absence of group-by and the fact that DB2 allows UDF in recursive queries mean that SQL-AG provides for general and efficient support of monotonic aggregates in SQL queries.

An interesting example is transitive ownership and control of corporations [15]. 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 corporation such as C2 that is controlled by C1. This problem can be solved by using a UDA **msum** that returns a new partial sum for each new value:

Example 3 *Transitive Control of Corporations in SQL and its DB2 implementation*

```
WITH control(C1, C2) AS
(   SELECT C1, C1
    FROM owns
UNION ALL
    SELECT c.C1, o.C2
    FROM control c, owns o
    WHERE c.C2 = o.C1
    GROUP BY c.C1, o.C2
    HAVING msum(o.Per)>49 )
SELECT C1, C2 FROM control
```

```
WITH control(C1, C2) AS
(   SELECT C1, C1
    FROM owns
UNION ALL
    SELECT c.C1, o.C2
    FROM control c, owns o
    WHERE c.C2 = o.C1
    AND msum_groupby(c.C1 || o.C2, o.Per)=0
    AND msum(c.C1 || o.C2) > 49 )
SELECT C1, C2 FROM control
```

In the next BoM example, we use sumat(K, X), which returns the sum as soon as K items are counted. Also, relation assembly(Part, Subpart, Qty) contains parts, their immediate subparts, and the quantity of subparts of the part. Relation part_cost(Part, Cost) stores the cost for each basic part. View prolfc(Part, ChC) stores the number of different subparts of each part. Then, to compute the cost for each part in the database we write:

Example 4 *BoM cost of parts in SQL-AG and its DB2 implementation*

```
WITH cst(Part, Cost) AS
(   SELECT Part, Cost
    FROM part_cost
UNION ALL
    SELECT a.Part, sumat(p.ChC, a.Qty × c.Cost)
    FROM assembly a, cst c, prolfc p
    WHERE a.Subpart = c.Part
    AND p.Part = c.Part
    GROUP BY a.Part
)
SELECT Part, Cost FROM cst
```

```
WITH cst(Part, Cost) AS
(   SELECT Part, Cost
    FROM part_cost
UNION ALL
    SELECT a.Part, sumat(a.Part)
    FROM assembly a, cst c, prolfc p
    WHERE a.Subpart = c.Part
    AND p.Part = c.Part
    AND sumat_groupby(a.Part, p.ChC,
          a.Qty × c.Cost) = 0 )
SELECT Part, Cost FROM cst
```

References

[1] Abiteboul S., Hull R. and Vianu V., Foundations of Databases, Addison Wesley, 1995.

[2] M. P. Consens and A. O. Mendelzon, “Low Complexity Aggregation in Graphlog and Datalog,” ICDT’90, 1990.

[3] I. F. Cruz and T. S. Norvell, “Aggregative Closure: an Extension of Transitive Closure,” Fifth Int. Conference on Data Engineering, pp. 384-389, 1989.

[4] Finkelstein, S. J., N.Mattos, I.S. Mumick, and H. Pirahesh, Expressing Recursive Queries in SQL, ISO WG3 report X3H2-96-075, March 1966.

[5] Punit Bhargava, “User-Defined Aggregates in Database Languages,” MS Thesis, UCLA CS 98, http://www.cs.ucla.edu/˜ czdemo/SQL-AG/ .

[6] 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.

[7] F. Gianotti, D. Pedreschi, and C. Zaniolo, “Semantics and Expressive Power of Non-Deterministic Constructs in Deductive Databases,” JCSS, to appear.

[8] Gyssen, M., Van Gucht, D. and Badia, A., Query Languages with Generalized Quantifiers, in Applications of Logic Databases, R. Ramakrishan, Kluwer, 1995.

[9] J. Gray, A. Bosworth, H. Pirahesh, A. Layman. “Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total.” *Proc. ICDE 1996.*

[10] J. M. Hellerstein, P. J. Haas, and H. J. Wang. “Online Aggregation.” In *Proceedings of ACM-SIGMOD Conference on Management of Data*, May 1997.

[11] Hsu, P. Y. and Parker, D. S., ”Improving SQL with Generalized Quantifiers,” Proc. ICDE 1995.

[12] Database Language SQL - Part 2: SQL/Foundation, July 1996.

[13] 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.

[14] I.S. Mumick, H. Pirahesh, and R. Ramakrishnan. The magic of duplicates and aggregates. In *Proceedings of the 16th Conference on Very Large Data Bases*, pages 264–277, 1990.

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

[16] 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.

[17] D. Saccà and C. Zaniolo, Deterministic and non-deterministic Stable Models, *Journal of Logic and Computation*, 7(5):555-579, October 1997.

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

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