New Generation Computing, 12 (1994) 131-160
OHMSHA, LTD. and Springer-Verlag

NEW GENERATION COMPUTING

© OHMSHA, LTD. 1994

Towards an Efficient Evaluation of Recursive Aggregates in Deductive Databases*

Alexandre LEFEBVRE
*BULL***
B.P. 68
78340 Les-Clayes-sous-Bois, France.

Received 16 November 1992
Revised manuscript received 1 October 1993

Abstract This paper is devoted to the evaluation of aggregates (*avg*, *sum*, ...) in deductive databases. Aggregates have proved to be a necessary modeling tool for a wide range of applications in non-deductive relational databases. They also appear to be important in connection with recursive rules, as shown by the *bill of materials* example. Several recent papers have studied the problem of semantics for aggregate programs. As in these papers, we distinguish between the classes of stratified (non-recursive) and recursive aggregate programs. For each of these two classes, the declarative semantics is recalled and an efficient evaluation algorithm is presented. The semantics and computation of aggregate programs in the recursive case are more complex: we rely on the notion of graph traversal to motivate the semantics and the evaluation method proposed. The algorithms presented here are integrated in the QSQ framework. Our work extends the recent work on aggregates by proposing an efficient algorithm in the recursive case. Recursive aggregates have been implemented in the EKS-V1 system.

Keywords: Deductive Database, Aggregate Function, Recursion, Extended Datalog, Database, Deduction

§1 Introduction

This paper examines an advanced functionality of deductive database systems, namely the ability to express programs involving both recursion and

---

* This paper is an extended version of the paper which appeared in the proceedings of the FGCS' 92 Conference.[21]

** This work was achieved while the author was at the European Computer-Industry Research Centre in Munich.

aggregate computations in a declarative manner. The *bill of materials* application (compute the total cost of a composite part built up recursively from basic components) shows the importance of this feature in real life databases. It is well known that such programs are not expressible in Datalog. We discuss a semantics, an evaluation model and the implementation of aggregates in the EKS-V1 system.[43]

The recursive aggregate facility is one of the innovative features of the declarative language of EKS-V1, in addition to the more standard features, recursion, negation and universal and existential quantification. EKS-V1 also provides an extensive integrity checking facility and sophisticated update primitives (hypothetical reasoning, conditional updates). EKS-V1 was mainly developed in 1989 and demonstrated at several database conferences (EDBT, Venice, March 1990; SIGMOD, Atlantic City, May 1990; ICLP, Paris, June 1991; VLDB, Barcelona, September 1991); it is used as a support for teaching in several universities.

The aggregate capabilities we consider are essentially those of SQL: a grouping primitive (*group_by*) is used in association with scalar functions (such as *sum*, *avg*, *min*) to compute aggregate values for each group of tuples. Adding aggregate capabilities to a recursive language causes different problems, depending on the class of programs accepted. We will consider two such classes: *stratified aggregate programs* and *non-stratified aggregate programs* (this terminology builds on an analogy with negation that will be explained below).

Our aim here is *to provide efficient evaluation algorithms* which can be integrated in a general evaluation framework such as QSQ or Magic Sets. In the case of EKS-V1, this is performed within the top-down QSQ/DedGin* framework which was developed in Refs. 44), 45), and 46) and for which compilation and implementation techniques were developed in a set-oriented way in the DedGin* prototype.[22] Studying evaluation in this framework does not limit its scope. Indeed, it is now accepted that there is a canonical mapping between an evaluation performed using a Magic Sets like strategy[28,5,37] and a "top-down" strategy[44,45,38] (see Refs. 9), 33), 40), and 46) for a comparison). Hence, anything that we develop here can be adapted to Magic Sets (and vice-versa).

In *stratified aggregate programs*, aggregate operations and recursion can't be interleaved. In other words, an aggregate operation may be specified over the result of a recursive query, or a recursive query may be specified over the result of an aggregate operation. However, an aggregate operation may not be part of a recursive cycle, i.e. one aggregate predicate can not recursively refer to itself.

For stratified aggregate programs, both semantics and evaluation issues are readily solved: 1) the semantics can be defined in a standard proof-theoretic way and 2) the evaluation problems are essentially those of top-down *constant propagation* and of *coordination* on the strata. The constant propagation issue is the (classical) problem of making use of constants given in the query to limit

the search space. For a query like "*Give me the average salary for the sales department*", one does not need to consult the entire employee relation. As for coordination, one has to make sure that all relevant tuples have been derived before performing the aggregate operation: again, this is a classical and relatively easy problem, which can be solved by appropriately extending the query evaluation method of the respective system.

In the case of *non-stratified aggregates*, interaction of recursion and aggregate computation raises more difficult problems. As a motivating example, consider the classical *bill of materials* application for a bicycle. In order to compute the total cost of a bicycle, one has to 1) compute the total costs of all its direct subparts (e.g. a wheel), 2) multiply these costs by the number of occurrences of these subparts (e.g. 2 wheels on a bicycle) and 3) sum up the resulting costs (aggregate computation). Step 1) consists of a recursive invocation of the *bill of materials* query, implying a recursive invocation of step 3) (aggregate computation). Clearly, aggregate computation and recursion are intertwined. In the following, we refer to this more general class of programs either as *non-stratified aggregate programs* or as *recursive aggregate programs*.

The first difficulty concerns semantics. For instance, suppose that, in the *bill of materials* example, a composite part is defined in terms of itself (cyclic data). Clearly, the cycle problem has to be solved in order to provide semantics for such queries. Our definition of the semantics of recursive aggregate queries relies on the two following intuitive choices. 1) We regard recursive aggregate computations as operations on top of the evaluation of a Datalog program. This underlying program represents a generalized graph (Datalog allows more than just transitive closure) being traversed during evaluation.[27] 2) Semantics should be definable in a way orthogonal to the semantics of the aggregate operations themselves: for example, the semantics of a query should be definable whenever *min* is replaced by *max* or vice-versa (of course, the result of the evaluation would be different!).

In order to give semantics to recursive aggregate programs, we consider the subclass of programs for which it is possible to associate a *reduced program* leaving out the associated computation of aggregates. This program conceptually represents the graph being traversed. We call such programs *reducible aggregate programs*. A query on a reducible program is meaningful only if there is no cycle in the derivations on the associated reduced program (we speak then of *group stratification*). Its semantics can then be defined in a classical proof-theoretic manner.

The second difficulty is the evaluation of recursive aggregate queries. As in the stratified aggregate case, this issue is two-fold: constant propagation and coordination. *Constant propagation* is done in the same way as in the stratified aggregate case. *Coordination* is more difficult than in the stratified aggregate case as one has to rely on *data stratification* (there is no predicate stratification any more). Hence, one has to ensure that the whole group of tuples for a *given input*

value has been derived before performing the corresponding aggregate operation. However, we are manipulating sets of tuples: in a given set of tuples at a given time, there might be a group that has been completely derived, and another for which only a partial set of tuples has been derived. This makes the control over the order of evaluation more complicated as it now has to be performed at the data level.

In the top-down evaluation scheme of EKS-V1, we introduce the notion of *subquery completion*. We rely on dependencies between subqueries in order to check whether the derivation of a given group has been completed. A general solution is proposed which makes use of the reduced associated program in order to provide ranges for the subqueries, so that the resulting subquery dependencies correspond to the group dependencies. In the case of tail-recursive programs, including the *bill of materials* program, a simplification is possible.

The main contribution of our work is the integration of recursion and aggregates in a general query evaluation framework. Two independent studies on recursive aggregates[23,11] have been developed in parallel to our work. They take a model-theoretic approach, whereas we consider a proof-theoretic approach to the semantics of aggregate programs. In Ref. 23), Mumick *et al.* describe an algorithm extending the Magic Sets technique to stratified aggregate programs (in fact *Magic Stratified* aggregate programs). In this paper, *we extend the evaluation algorithm based on QSQ to group stratified aggregate programs* of which the *bill of materials* program is an example.

More recently, a technique similar to ours called Ordered_Search[31] has been developed and implemented in the Coral deductive database system.[32]

The structure of this paper is as follows. The remainder of this section introduces some definitions and notations. Section 2 examines semantics and evaluation of stratified aggregates. For the recursive aggregate case, we first analyze the semantics problem in Section 3, where we define the class of reducible aggregate programs. We then propose an evaluation method which relies on the notion of subquery completion in Section 4. Section 5 discusses related work. Section 6 summarizes the paper and describes future work.

1.1 Definitions and Notations

We assume that a database is composed of base relations and of deduction rules of the form *Head* ← *Body* where the *Body* is a conjunction of positive and negative literals. All the variables in the *Head* should appear in a positive literal in the body. Deduction rules define virtual predicates, which are also commonly called *views* in the classical relational terminology.

Definition 1.1 Aggregate rule

An **aggregate predicate** *agg_pred* is syntactically defined, as in Ref. 23), by an **aggregate rule** in the following way:

agg_pred(Oūt) ← **group_by**(

group_pred($I\vec{n}$),
List_of_Grouping_Variables,
List_of_Aggspecs
).

where:

- *List_of_Grouping_Variables* is a list of variables. $O\vec{u}t$ and $I\vec{n}$ are sequences of variables. They are called **grouping, output and input variables** respectively;
- *group_pred* is any virtual or base predicate and is called the **grouping predicate**;
- *List_of_Aggspecs* is a list of aggregate specifications of the form A **isagg** *func_agg*(B) or A **isagg** *count* where *func_agg* can be 'sum', 'min', 'max' or 'avg', A must be an output variable and B must be an input variable. The variable A is called an **aggregate variable** and B a **variable to-be-aggregated**;
- an output variable must either be a grouping variable or an aggregate variable.

Without loss of generality, we assume that an aggregate predicate is defined by one aggregate rule only. □

Note that the aggregate function *count* has no argument, as it simply counts the number of tuples for a given group.

We allow the use of arithmetic predicates in the body of Datalog rules. Such predicates, not computable by the basic relational operations, are called *external predicates*. We suppose that the external predicates are used in a *safe* way (as in Ref. 10)—there should be a finite set of answers and finite top-down evaluation). As an example, the *bill of material* example uses an external predicate performing a multiplication (see Section 3). The use of this predicate is safe as long as the data is acyclic.

Comparing our syntax to the SQL syntax for aggregates deserves two remarks. First, SQL allows the user to specify aggregate functions on any query, including joins and any kind of *where* condition. With our syntax, however, aggregate functions can only be applied to a single grouping literal. A natural extension would be to allow the user to write any expression, instead of a single grouping literal. Second, SQL allows the user to specify conditions on the result of the aggregations by means of the *having* clause inside an aggregate query. This is also possible with our syntax: the corresponding condition however has to appear outside the aggregate rule, because it concerns the *result* of the aggregation and not the aggregate computation itself. We feel that it is a more natural way to express such conditions.

Definition 1.2 Grouping subtuples and groups of tuples
Given a tuple for the grouping predicate, its **grouping subtuple** is its projection

over the grouping arguments.
Given a set of tuples S for a grouping predicate, we partition S into *groups of tuples*: there is one **group** for each different grouping subtuple GST in S. A group contains those and only those tuples of S having GST as grouping subtuple (and no other tuple). □

We say that a predicate $pred_1$ depends directly (resp. indirectly) on the predicate $pred_2$, if $pred_2$ appears in the body of a rule defining $pred_1$ (resp. if there is a predicate $pred_3$ such that $pred_1$ depends directly on $pred_3$ and $pred_3$ depends indirectly on $pred_2$). We can now give the following definition, inspired by the terminology used in the case of Datalog queries with negation.

Definition 1.3 Stratified aggregate program
An aggregate program is **stratified** if no aggregate predicate depends directly nor indirectly on itself. □

Note that aggregate programs having recursive predicates which are not mutually recursive with aggregate predicates are indeed aggregate stratified.

A simple example of a stratified aggregate program is the following.

Example 1.1
Suppose that the database contains a base relation *employee* with tuples of the form *employee(Name, Dept, Salary)*. One can define a virtual predicate *avg_salary_per_dept* using the following rule:

```
avg_salary_per_dept(Dept, AvgSal) <-
        group_by(
                employee(Name, Dept, Salary),
                [Dept],
                [AvgSal isagg avg(Salary)]
              ).
```

If the predicate *avg_salary_per_dept* is queried with the argument *Dept* instantiated, it returns one single value. If the query is fully uninstantiated, the result is a binary table with one value per department. □

§2 Stratified Aggregates

In this section, we first recall the natural semantics of stratified aggregate programs, which rely on the stratification of rules. We then describe their evaluation by extending the QSQ framework.

2.1 Semantics

The stratification of a database ensures the soundness of the following extension of the classical proof-theoretic definition of semantics for stratified aggregate programs.

Like Datalog programs with stratified negation, a stratified aggregate

program P can be divided into *strata* S_i, $i = 1, ..., n$.

Consider a predicate p appearing in the body of a rule $R \in S_i$. If R is an aggregate rule and p appears as a grouping predicate in R, then the definition of p is contained in $\bigcup_{j<i} S_j$, else, its definition is contained in $\bigcup_{j\leq i} S_j$.

An example of a stratified aggregate program involving more than one aggregate predicate is the following, counting the number of descendants for each person in a family tree.

Example 2.1

Suppose that the database contains a base relation *child* with tuples of the form *child*(X, Y) where Y is a child of X. One can define the virtual predicate *children_nr*, counting the number of children of a given person, with the following rule:

```
children_nr(X, N) <-
        group_by(
                child(X, Y),
                [X],
                [N isagg count]
               ).
```

The *descendant* predicate, transitive closure of the *child* predicate, is defined with the following rules:

```
descendant(X, Y) <-
        child(X, Y).
descendant(X, Y) <-
        child(X, Z) and descendant(Z, Y).
```

Building on these two predicates, the predicate *children_nr_for_descendant* computes the number N of children of the descendant Y of a person X as follows:

```
children_nr_for_descendant(X, Y, N) <-
        descendant(X, Y) and children_nr(Y, N).
```

Finally, the predicate *descendant_nr*, representing the number of descendants for each person, is defined with the following rule:

```
descendant_nr(X, N) <-
        group_by(
                children_nr_for_descendant(X, Y, N),
                [X],
                [N isagg count]
               ).
```

This program is aggregate stratified. No aggregate predicate depends directly or indirectly on itself. A possible stratification for this program is as follows: $S_1 =$

$\{child, descendant\}$, $S_2 = \{children_nr, children_nr_for_descendant\}$ and $S_3 = \{descendant_nr\}$. □

Definition 2.1 Semantics of a stratified aggregate program *P*
Facts derivable for *P* from the database are obtained by saturation of the immediate consequence operator, consecutively on each stratum S_i, starting from $i = 1$ up to $i = n$. Facts for aggregate predicates are defined as follows. For an aggregate predicate *agg_pred*, there is one tuple T_G for each group *G* of the corresponding grouping predicate *group_pred* such that:

(1) *If an attribute of* T_G *corresponds to a grouping variable*, its value is the value of the same variable in *G*.
(2) *If an attribute of* T_G *corresponds to an aggregate variable*, its value is the result of the aggregate operation performed on the corresponding values of *G* to be aggregated. □

Note that this proof-theoretic definition of the semantics is equivalent to the model-theoretic one given in Refs. 23) and 11). For stratified aggregates, Mumick *et al.*[23)] define the model-theoretic semantics in terms of a perfect model. The proof-theoretic definition of the semantics defined above exactly computes this perfect model (it is similar to the construction of the perfect model of stratified programs with negation).

2.2 Evaluation

We present here an evaluation algorithm integrated in the QSQ framework. In Ref. 23), the authors extend the Magic Set formalism to stratified aggregates in a similar way.

(1) Constant propagation

The propagation of constants (i.e. taking advantage of the constants appearing in the query in order to reduce the search of the database) is addressed by adapting the QSQ framework: the top-down generation of subqueries is used for focusing on relevant data while answers are propagated bottom-up.

We first describe this adaptation on a tuple-at-a-time basis. Let *Q* be a query over the aggregate predicate *agg_pred* defined using an aggregate rule as in Definition 1.1. Answering *Q* consists in the following steps:

(1) If *Q* matches the head *agg_pred*(*Out*) of the aggregate rule, then generate a subquery *SQ* on *group_pred* by binding each variable *X* of *group_pred* which is also present in *agg_pred* (*X* must be a grouping variable) to its value in *Q* (either a variable or a constant).
(2) Answer the subquery *SQ*.
(3) Partition the answers to *SQ* into groups of tuples and perform the aggregate operations for each group.
(4) Project the results over the arguments of *agg_pred*.

Note that only the bindings of grouping variables are propagated downwards. If some aggregate variables of *agg_pred* are bound in *SQ*, then their bindings are not propagated to *SQ* (e.g. if a value for the *AvgSal* argument of Example 1.1 is provided in the query, then this binding is not propagated). The gain obtained by using such bindings in order to reduce the search space depends on the nature of the aggregate and can require a complicated mechanism.

(2) Set-oriented evaluation in EKS-V1

The evaluator/compiler of EKS-V1 derives from the DedGin* prototype. The above computational scheme is implemented in a set-oriented way by a simple adaptation of the DedGin* query answering mechanism. The following operations correspond to the previously described steps:

(1) A selection/projection selects from a set of queries *Q* those queries matching the head of the aggregate rule, and projects the resulting tuples over the relevant arguments of *group_pred*. This results in a set of subqueries *SQ* over *group_pred*.
(2) The standard set-oriented evaluation of DedGin* is used to answer the subqueries in *SQ*.
(3) The grouping and aggregate operations are implemented in one pass, by an extended operator described below. This results in an intermediate relation *tmp* containing one attribute for each grouping variable and for each aggregate variable.
(4) A projection of the tuples in *tmp* over answer tuples for *agg_pred* is finally performed.

(3) Remarks

In EKS-V1, only step (3) needs an extension to the basic set-oriented machinery: grouping and aggregate operations are performed by one single set-oriented operator implemented in BANG.* This operator projects the grouping tuples onto their grouping value (grouping operation) and incrementally performs the aggregate operations on additional arguments associated with each grouping subtuple. In the case of an average, the *sum* and *count* aggregates are computed during step (3), and then a division during step (4).

The 4th operation is needed when some grouping variables do not appear in *agg_pred*. In such cases, it is crucial that the aggregate operations are first performed on the *full grouping subtuples* (resulting in the *tmp* relation) and then the intermediate result *tmp* be projected out to the final answer relation. As an example, one may want to compute the different values of the average salary per department *without the values of the departments*. For this purpose, the

* The BANG file system[14] provides the data manipulation operators in EKS-V1.

variable *Dept* would be in the grouping list but would not appear in the head, i.e. in the aggregate predicate.

(4) Coordination aspects

In general, the evaluation of deductive queries can be viewed as a saturation both on the top-down propagation of (non-redundant) subqueries and on the bottom-up generation of answers. In the case of recursion without negation or aggregates, there is total freedom as far as the order of propagation is concerned. In particular, answers can be propagated bottom-up even if they represent only a partial set of answers to the corresponding subqueries. However in case of aggregates (also in the case of negation), *subqueries must be answered completely before their answers can be used or propagated further*. If one did not stick to this strategy, wrong inferences could be made: for instance, one could propagate an intermediate count which is different from the final count.

In order to implement this strategy in EKS-V1, we make use of a run-time structure described in Refs. 45) and 22) called the *data-flow graph* (DFG). Nodes of this graph essentially represent (occurrences of) virtual predicates and the graph serves to monitor the sets of data (essentially subqueries, environments and answers) manipulated for these (occurrences of) predicates. The nodes are linked according to their relative positions in rules: the *brother* of a node corresponds to the immediate next literal in the body of a rule; predicates in the body of a rule defining a virtual predicate *p* form *children* nodes with respect to the node corresponding to *p*. Refer to Refs. 45) and 22) for a precise definition of the DFG. This structure is adequate for coordination aspects since it gives, at any time, a "map" of the rules that have been evaluated or remain to be evaluated to fully answer a virtual predicate. The coordination strategy described above can be formulated in the case of aggregate predicates as follows:

> For each node *N* of the DFG corresponding to an aggregate predicate, saturate the descendants of *N* before performing the aggregate operation associated with *N*.

§3 Semantics of Recursive Aggregates

In order to introduce problems arising in the case of recursive aggregate programs, we discuss the classical *bill of materials* example, also presented in Refs. 23) and 11).

Example 3.1 Bill of materials

Suppose that the database contains the following information: *basic parts*, and their cost and *assembly links*, to make up composite parts are stored in two base relations

```
basic_part(Part, Cost).
assembly(Part, SubPart, Qty).
```

The *bom* (bill of materials) predicate computes the total cost of a given part by summing up the costs of all its direct subparts, computed by the grouping predicate *subpart_cost*.

```
bom(Part, TotalCost) <-
        group_by(
                 subpart_cost(Part, SubPart, Cost),
                 [Part],
                 [TotalCost isagg sum(Cost)]
                ).
```

The non-recursive rule of *subpart_cost* returns the cost for a basic part. The recursive rule computes the cost which a direct subpart *SubPart* accounts for in the total cost of *Part* by recursively computing its cost and multiplying it by the number of occurrences of *SubPart* in *Part*.

```
subpart_cost(Part, Part, Cost) <-
        basic_part(Part, Cost).
subpart_cost(Part, SubPart, Cost) <-
        assembly(Part, SubPart, Quantity)
        and bom(SubPart, TotalSubCost)
        and Cost is Quantity * TotalSubCost.
```

As an example, if *Part* is "bicycle", and if "bicycle" is made up of two wheels (each costing 10) and of one frame (costing 100), then the subquery *subpart_-cost*(*bicycle*, *Subpart*, *Cost*) will return two tuples:

```
(wheel, 20)       % 20 is  2 * 10
(frame, 100)      % 100 is 1 * 100
```

The aggregate computation performed in the rule defining *bom* then returns 120 as the total cost for a "bicycle". □

What would the semantics of the *bill of materials* example be if there were a cycle in the data: what would be the cost of a recursively defined composite part (where its value depends on itself)? In order to solve this problem, we rely on the following two choices:

(1) We intuitively view recursive aggregate computations as *generalized graph traversals*. In this framework, computations are performed both along deduction paths (e.g. multiplying by the number of occurrences) and by aggregating the values associated with several paths (summing up costs). However, recursive aggregate computations go beyond graph traversal as they require 1) more complex structures than graphs to be searched (n-ary relations correspond to hypergraphs), 2) the combination of several "graphs" in the search (several, different predicates) and 3) a more general search than transitive closure (e.g. non-linear recursion).

To each recursive aggregate program, we conceptually associate a so-called *reduced program*. Intuitively, the reduced program captures the essence of traversal, while leaving out the associated computation of aggregates. We provide a rewriting method which, given a recursive aggregate program, obtains its reduced program if one exists.

A recursive aggregate program is then acceptable if it is syntactically correct, i.e. if there exists a reduced program attached to the original aggregate program. In such a case, the program is said to be *reducible*.

(2) Moreover, we consider that the semantics should be definable in a way orthogonal to the semantics of the aggregate operations: for example, the cases where the semantics of a query is defined should be the same whenever *min* is replaced by *max* or vice-versa (however, the result of the evaluation would be different). As a consequence, we give semantics to recursive aggregates only when *the data is acyclic*, i.e. if the proof trees generated from the database for the reduced query are *acyclic*. The actual semantics of meaningful recursive aggregates queries is then defined in a classical bottom-up manner.

Indeed, although one could compute the shortest path between two nodes of a cyclic graph, one can not compute the *maximal* length of a path in such a case. However accepting the first case without accepting the second one would violate this orthogonality principle.

3.1 Reducible Aggregate Programs and Group Stratification

We conclude the semantics chapter by giving more precise definitions of the notions "reduced", "reducible" and "acyclic" introduced above.

Consider the program P consisting of the set of rules defining the predicates which are mutually recursive with a given aggregate predicate *agg_-pred*.

We built the *variable graph* $\mathcal{V}$ for P as follows. There is one node (p, i) in $\mathcal{V}$ for each variable position i of each virtual predicate p in P. There is an edge between two nodes in $\mathcal{V}$ if there is a rule r in P such that the variables corresponding to the nodes appear in the same external predicate in the body of r. There is an edge between two nodes (p, i) and (q, j) in $\mathcal{V}$ if there is a rule in P of which p is the head predicate, q is a body predicate, and the variables corresponding to i and j are identical. A node (p, i) is an **aggregate node** if there is a rule r in P of which either p is a grouping predicate and the corresponding variable appears as a variable to-be-aggregated in the body of r, or p is the head predicate and the corresponding variable appears as an aggregate variable in the body of r. Finally, a node (p, i) is a **grouping node** if there is an aggregate rule r in P of which p is the head predicate and the corresponding variable appears as a grouping variable in the body of P.

Obtaining a reduced program from an original program P will be possible if the grouping variables, which represent the essence of the program,

can be isolated from the aggregate variables: the program P is *reducible* if no grouping node is connected to an aggregate node in $\mathcal{V}$.

In a given rule r of P, a variable is said to be **aggregate connected** if the node in $\mathcal{V}$ corresponding to its position is connected to an aggregate node. Informally, if P is reducible, its reduced program *reduce*(P) is obtained by 1) deleting from each rule of P every literal, built on an external predicate, which contains aggregate connected variables and 2) replacing each literal which is mutually recursive with *agg_pred* by a new literal where the aggregate connected variables have been omitted (hence, reducing its arity). Indeed, if P was not reducible, then the transformation *reduce* would also remove some grouping variables which carry the essence of the program.

Formally, the transformation *reduce* makes use of a set *R_Set* (initialized to $\emptyset$) containing pairs of the form (*Lit*, *R_Lit*), where *Lit* is an initial literal and *R_Lit* is obtained from *Lit* by omitting some variables. The predicates of the reduced literals are renamed by adding the prefix "*r_*" in front of the initial predicate names.

Definition 3.1 Transformation *reduce* and *reducible* programs
The transformation reduce is defined as follows:

(1) Replace each aggregate rule:

```
Head ← group_by(
                Group_Lit,
                List_of_Grouping_Variables,
                List_of_Aggspecs
               ).
```

by:

```
Head′ ← Group_Lit′.
```

where *Head′* (resp. *Group_Lit′*) is obtained from *Head* (resp. from *Group_Lit*) by deleting the aggregate variables (resp. the to-be-aggregated variables).
Add (*Head*, *Head′*) and (*Group_Lit*, *Group_Lit′*) to *R_Set* while replacing all arguments by new variables.

(2) Iterate the following process until no more changes occur:
Replace each remaining rule in the program (these are not aggregate rules): *Head* ← *Body* by a new rule: *Head′* ← *Body′* obtained by:

(a) replacing each literal *Lit* (including the head) by its corresponding *R_Lit* whenever (*Lit*, *R_Lit*) is in *R_Set*;
(b) removing all external predicates of which one of the input arguments corresponds to one of the variables removed in step (a);
(c) if step (b) was applied, obtaining *Head′* from *Head* (or from

R_Head if step (a) was applied) by removing the output variables of the external predicates involved in step (b). In this case, add (*Head*, *Head'*) to *R_Set* (replace all arguments by new variables).

The transformation *reduce* succeeds and the recursive aggregate program *P* is said to be reducible if the saturation process in (2) does not affect the grouping and aggregate literals. □

From now on, we consider only reducible aggregate programs.*

In order to illustrate the concepts defined here, let us introduce the *parts explosion* example, which computes the total amount *Qty* of a given subpart *SP* involved in the construction of a given part *P*. The definition of *part_subpart_-qty* has the same structure as the definition of *bom*. It uses a grouping predicate *int_subpart_qty* which gives, for each direct intermediate component *IP* of *P*, the quantity of *SP* involved through *IP*. Note that the predicate *part_subpart_-qty* is an extension of the *bom* predicate with more didactic properties.

Example 3.2 Parts explosion and reduced program

```
part_subpart_qty(P, SP, Qty) <-
        group_by(
                int_subpart_qty(P, IP, SP, IQty),
                [P, SP],
                [Qty isagg sum(IQty)]
              ).

int_subpart_qty(P, P, SP, Qty) <-
        assembly(P, SP, Qty).

int_subpart_qty(P, IP, SP, IQty) <-
        assembly(P, IP, Qty) and
        part_subpart_qty(IP, SP, IQty1) and
        IQty is Qty * IQty1.
```

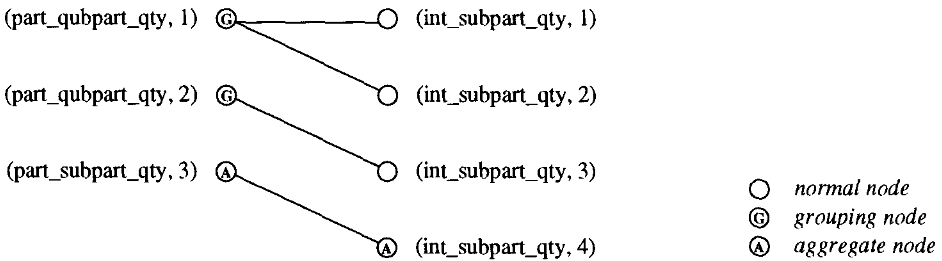

Fig. 1 Variable graph for the parts explosion program.

* In practice, the only reasonable recursive aggregate programs we could think of are reducible (bill of materials, shortest path, corporate takeover, ...). This is also the case of all examples treated in the related work.

Figure 1 represents the variable graph for this program. There is an edge between (*part_subpart_qty*, *3*) and (*int_subpart_qty*, *4*), because the corresponding variables appear in the external multiplication predicate. There is an edge between (*part_subpart_qty*, *1*) and (*int_subpart_qty*, *1*), and an edge between (*part_subpart_qty*, *2*) and (*int_subpart_qty*, *3*) because the corresponding variables are identical in the aggregate rule defining *part_subpart_qty*. Finally, there is an edge between (*part_subpart_qty*, *1*) and (*int_subpart_qty*, *2*) because the corresponding variables are indentical in the third rule. The aggregate nodes in the variable graph corresponding to this program are (*part_subpart_qty*, *3*) and (*int_subpart_qty*, *4*); the grouping nodes are (*part_subpart_qty*, *1*) and (*part_subpart_qty*, *2*). No grouping node is connected to an aggregate node, therefore the program is reducible. The reduced program is:

```
r_part_subpart_qty(P, SP) <-
        r_int_subpart_qty(P, IP, SP).

r_int_subpart_qty(P, P, SP) <-
        assembly(P, SP, Qty).
r_int_subpart_qty(P, IP, SP) <-
        assembly(P, IP, Qty) and
        r_part_subpart_qty(IP, SP).
```

□

We now define precisely what we mean by "cyclic data".

Definition 3.2 Fact and group dependencies
A fact F derivable from DB is **directly dependent** on a fact F' if there is a ground instance I of a clause such as I: $F \leftarrow$ ... *and* F' *and* ... and such that all the ground literals of the body of I are derivable from DB. The dependency relationship is the transitive closure of the direct dependency relationship.
The **group dependency** relationship is the fact dependency relationship induced by $reduce(P)$ over DB. □

Definition 3.3 Group stratified program
A recursive aggregate program P is **group stratified** over a database DB if the group dependency relationship introduced by P over DB is acyclic. □

Example 3.2 (continued) Consider the following data for the assembly relation representing a bicycle, as illustrated in Fig. 2:

```
assembly(bicycle, frontframe, 1).
assembly(frontframe, mudguard, 1).
assembly(frontframe, fork, 1).
assembly(frontframe, wheel, 1).
assembly(frontframe, headlamp, 2).
assembly(frontframe, handles, 1).
assembly(bicycle, rearframe, 1).
assembly(rearframe, mudguard, 1).
assembly(rearframe, fork, 1).
assembly(rearframe, wheel, 1).
assembly(rearframe, chain, 1).
assembly(rearframe, pedal, 1).
assembly(rearframe, saddle, 1).
assembly(rearframe, reflector, 1).
```

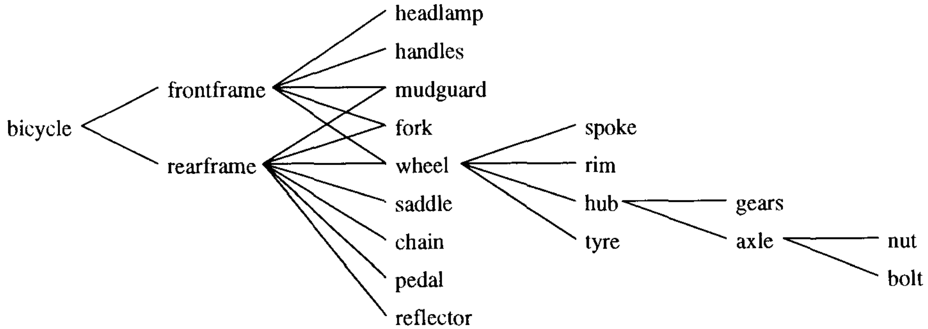

Fig. 2 Assembly graph of a bicycle.

```
assembly(wheel, hub, 1).          assembly(hub, gears, 1).
assembly(wheel, tyre, 1).         assembly(hub, axle, 1).
assembly(wheel, spoke, 20).       assembly(axle, nut, 2).
assembly(wheel, rim, 1).          assembly(axle, bolt, 1).
```

The parts explosion program together with these facts is clearly group stratified, as there are no cycles in the data. □

We can now define the semantics of a group stratified program P over DB, by refining Definition 2.1. Again, the notion of group stratified programs here is identical to the one proposed in Ref. 23).

This time, we note that the facts in *reduce*(P) can be divided along *group strata* GS_i, $i = 1, ..., n$, such that, if a fact $F_i \in GS_i$ depends on a fact $F_j \in GS_j$, then $j < i$. In addition, grouping and aggregate facts in P will be given the group stratum level of the corresponding reduced facts.

Definition 3.4 Semantics of a group stratified aggregate program P
Facts derivable for P from the database are obtained by saturation of the immediate consequence operator consecutively using facts belonging to the group strata $GS_{j \leq i}$, starting from $i = 1$ up to $i = n$. Facts for aggregate predicates are derived as in Definition 2.1. □

§4 Evaluation of Reducible Group Stratified Aggregate Programs

The evaluation problems in the recursive aggregate case are, like in the stratified aggregate case, those of constant propagation and coordination. As far as *constant propagation* is concerned, the problem is solved in the recursive aggregate case as described in Section 2.2 (1).

The *coordination* problem is now different. The goal is still to perform the aggregate operations only on *complete groups*. However, there is no predicate stratification in the recursive case, and a control mechanism as described in Section 2.2 (4) cannot be performed any more. Instead, the group stratification that the program is supposed to enforce is data dependent and not predicate

dependent. Hence, the coordination will have to be performed at the data level instead of at the predicate level. In Ref. 23), Mumick *et al.* remark that group stratified programs can be evaluated in the order of the groups. In this section we give a precise algorithm to perform this evaluation.

Theoretically one could first generate the group dependency graph and base the computation on this graph. However, the representation and analysis of such a graph is likely to be expensive.

The solution proposed in EKS-V1 relies on the top-down character of the evaluation: there exist natural dependencies between the *subqueries*. A subquery *SQ* is said to **directly depend** on the subqueries derived during the evaluation of the rules invoked for answering *SQ*; a formal description of these dependencies can be provided based on SLD-AL trees—see Ref. 45). In Section 4.1 we present the *subquery completion* mechanism: the evaluation of a program under subquery completion ensures that *the set of answer tuples to a subquery is propagated only when it is complete*. In Section 4.2 we apply this technique to recursive aggregates. Modification of the original program using reduced literals is proposed in order to establish a one-to-one correspondence between subquery dependencies and group dependencies. The subquery completion mechanism can then be applied to the modified program. Section 4.3 is concerned with tail-recursive rules: in such a case, the subquery dependencies naturally correspond to the group dependencies and the original program can directly be evaluated under subquery completion.

4.1 Subquery Completion

We consider that a subquery has been completed during evaluation if its *complete* set of answers has been generated.

Definition 4.1 Subquery completion

A given subquery *SQ* has been **completed** if one of the two following conditions holds:

- *for a subquery on a base predicate*: the join with the corresponding base relation has been performed;
- *for a subquery on a virtual predicate*: all the rules have been fired, and recursively all the subqueries on which *SQ* directly depends have been completed.

We say that a program is evaluated under **subquery completion** if the set of answers to each subquery *SQ* is propagated only when *SQ* has been completed.

□

The subquery completion mechanism can be implemented as follows:

(1) When a subquery is derived, it is originally marked as non-completed.
(2) When answering a set of subqueries for which all the rules have been

triggered, the subqueries having non-completed direct descendants are left out. The other subqueries are marked as completed and the join with their corresponding answer tuples can take place.

4.2 Evaluation with the Reduced Program

Our goal is now to use the subquery completion mechanism in order to solve the problem of recursive aggregate evaluation. However, the subquery completion mechanism ensures that answers to a *subquery* are used when it has been completed, but not when a given *group* of tuples has been completed. We use calls to the reduced program in order to generate bindings for the grouping variables: this way, all grouping variables are instantiated and *the subquery tuples are identical to the grouping subtuples*. It follows that the subquery dependencies and the group dependencies coincide.

For reducible aggregate programs, the definition of a completed subquery is extended in the following way.

Definition 4.1 Subquery completion (continued)

- *a subquery on a grouping literal has been completed if* the aggregate operation on the corresponding group has been performed.

A reducible aggregate program is evaluated under **subquery completion** if the set of answers to each subquery SQ is propagated only when SQ has been completed, *and* if the aggregate operation on a given group is performed only when the corresponding subquery on the grouping predicate has been completed. □

Consider a recursive aggregate program P. The algorithm can be formalized as follows.

Algorithm 4.1

(1) Produce the corresponding reduced program $reduce(P)$.
(2) Modify P by introducing, at the beginning of each rule's body containing grouping and/or aggregate literals, the corresponding reduced literals. The evaluation of these reduced literals will be performed before the evaluation of the other literals and will provide bindings for all the grouping variables. Let P' be the resulting program.
(3) Modify the query by adding the corresponding reduced literal.
(4) Evaluate the modified query under subquery completion over $reduce(P) \cup P'$.

Thanks to the instantiations of all the grouping arguments by the reduced literals, the *subquery dependencies* correspond exactly to the *group dependencies*: the completion mechanism applied to the modified program guarantees that a given group is used for aggregate operations only when it is complete.

Example 4.1 (Example 3.2 continued)
Consider a query *part_subpart_qty(P, *SP, Qty)* (where "*" marks an argument which is instantiated when the literal is consulted during evaluation). Suppose that the compiler chooses the following ordering of the subqueries for the recursive rule of *int_subpart_qty*.

```
int_subpart_qty(P, IP, *SP, IQty) <-
        part_subpart_qty(IP, *SP, IQty1) and
        assembly(P, *IP, Qty) and
        IQty is *Qty * *IQty1.
```

The evaluation of the recursive rule for *int_subpart_qty* immediately generates subqueries on *part_subpart_qty* which are redundant w.r.t. the initial query on *part_subpart_qty*: they have the same argument **SP* carrying the same value. This introduces a cycle in the subquery dependencies. However, the group dependencies are cycle free for this example as long as the relation *assembly* is not cyclic.

Using the reduced literals for generating bindings for the grouping variables has the following effect on our example. The call to the query literal is replaced by "*r_part_subpart_qty(P, *SP)* and *part_subpart_qty(*P, *SP, Qty)*". The modified version of the program is:

```
part_subpart_qty(P, *SP, Qty) <-
        r_int_subpart_qty(P, IP, *SP) and
        group_by(
                int_subpart_qty(*P, *IP, *SP, IQty),
                [*P, *SP],
                [Qty isagg sum(IQty)]
              ).

int_subpart_qty(*P, *P, *SP, Qty) <-
        assembly(*P, *SP, Qty).

int_subpart_qty(*P, *IP, *SP, IQty)<-
        assembly(*P, *IP, Qty) and
        r_part_subpart_qty(*IP, *SP) and
        part_subpart_qty(*IP, *SP, IQty1) and
        IQty is *Qty * *IQty1.
```

The subquery dependencies on the modified program now correspond to the group dependencies.
Note as well that the reduced literal *r_part_subpart_qty(*IP, *SP)* in the recursive rule is superfluous as the two grouping arguments **IP* and **SP* would have been instantiated anyway. It can be removed. □

[1] Remarks

(1) When evaluating a program using the QSQ mechanism, each subquery is evaluated only once. A subquery SQ', which is a variant of a previously encountered subquery SQ, is called a *non-admissible subquery*. A non-admissible subquery SQ' is not reevaluated. Instead, the answers to the corresponding original subquery SQ are reused for answering SQ'. In order to take non-admissible subqueries into account, we have to extend the definition of completed subquery as follows.

Definition 4.1 Subquery completion (continued)

a non-admissible subquery has been completed if *the corresponding admissible subquery has been completed*.

The subquery completion mechanism can be extended with the following condition in order to handle non-admissible subqueries:

(3) When an admissible subquery has been completed, the corresponding non-admissible subqueries are marked as completed as well. □

This way, like during normal QSQ evaluation, an aggregate subquery is answered only once, and the corresponding answers are reused for variant occurrences of a given subquery. Consider again the parts explosion Example 3. 2 with the *bicycle* data. During the execution of the top query *part_subpart qty*(*bicycle*, P, Q), the subquery *part_subpart_qty*(*wheel*, P', Q') is derived twice, once through the front frame and once through the rear frame. Only one of these two subqueries will be evaluated. Once the evaluation of one occurrence of *part_subpart_qty*(*wheel*, P', Q') is completed, the answers are reused for the other occurrence of this subquery.

(2) When using the QSQ mechanism, it is important that the subquery completion is local to the recursive aggregate cliques. Consider that a recursive aggregate predicate *ap* depends on a recursive (non-aggregate) predicate *rp* which is in a lower clique, and that the completion mechanism is used on the whole program. Consider as well that the program is group stratified. Finally, suppose that the evaluation of some subquery on *rp* is such that it depends on itself (like in the *ancestor* example with a query *anc*(X, a)). If the subquery completion mechanism was applied to the predicate *rp* as well, then the non-admissible subquery for *rp* would never be completed, and no answer would be returned to the query on *ap*.

A correct evaluation scheme is achieved within the EKS-V1 system by applying the subquery completion mechanism only locally, within each recursive aggregate clique, and not within lower cliques.

(3) Note that the evaluation of reducible aggregate programs which are *not* group stratified stops, and returns a negative answer. As there are cycles

in the dependencies, there exists a subquery which depends on itself. This subquery will never be completed and the evaluation stops.

(4) One can emphasize the difference with the problem of handling negation in the effectively stratified case.[2,26,8] In programs with negation, the body of a rule always provides a range for all the variables of negated literals. Therefore there is no need to introduce a range in the case of negation. For recursive aggregate programs, reduced literals have to be introduced in the original program in order to obtain a range for all the grouping variables.

4.3 Simplification in the Tail-Recursive Case

The mechanism we have just presented has a drawback. For the evaluation of a query on an aggregate predicate the evaluator performs the search through the relevant data twice: once during the evaluation of the reduced predicates, and once during aggregate computation. There is a case however where the subquery dependencies naturally correspond to the group dependencies, even though some of the grouping arguments can be uninstantiated in the subqueries. In such a case, it is sufficient to evaluate the original aggregate program under subquery completion, therefore searching the data only once.

This case has been called *tail-recursive* in Ref. 22), and also corresponds to the *right- and left-linear recursive case* as in Ref. 24). A tail-recursive program is characterized by the following property.

Definition 4.2 Tail-recursive program

Consider the program *P* defining a recursive predicate *p* and a given query literal *QLit* built on predicate *p*. Consider that the literals in the rules defining *P* have been reordered for *QLit*. Consider each body literal *BLit* built on *p* appearing in the body of a rule *R* in *P*. *P* is **tail-recursive** w.r.t. *p* for *QLit* if and only if, for each *BLit*:

(1) *BLit* is the only literal in the body of *R* recursive with *p*.
(2) *BLit* is the last literal in the body of *R*.
(3) The variables which are different in *BLit* and *QLit* are instantiated.
(4) The variables which are free in *BLit* and *QLit* are the same and at the same position in those literals. □

By extension, we say that **a recursive aggregate program is tail-recursive** if its reduced program is tail-recursive w.r.t. the aggregate predicate.

We now give the proof of why a tail-recursive aggregate program *P* w.r.t. a given query literal *QHead* can be evaluated directly under subquery completion.

Proof 4.1

We have to guarantee that the subquery completion mechanism stops *exactly* when there exists a cycle in the group dependencies. This means that we have to

prove that a cycle in the group dependencies implies a cycle in the subquery dependencies (i) and vice versa (ii).

(i) Suppose that there is a cycle in the group dependencies. As the subquery tuples on the aggregate predicate are subtuples of the grouping tuples (remember that we do not propagate instantiations of the aggregate variables), a cycle in the groups dependencies implies a cycle in the subquery dependencies on the aggregate predicate.

(ii) Suppose that there is a cycle in the subquery dependencies on the aggregate predicate: a subquery *SQ1* depends on a variant subquery *SQ2* (note that we consider only cycles in the subquery dependencies for the aggregate predicate: because the aggregate predicate and the grouping predicate are mutually recursive, this also implies cycles in the subquery dependencies for the grouping predicate). Let us call *SQG1* (resp. *SQG2*) the subquery tuple on the grouping predicate on which *SQ1* (resp. *SQ2*) directly depends. *SQG1* and *SQG2* are variants as well. Suppose that *TG2* is an answer tuple to *SQG2*. As *SQG1* and *SQG2* are identical, *TG2* is also an answer tuple to *SQG1*. Moreover, as the reduced program is tail-recursive, the subqueries on the aggregate predicate share the same free grouping variables. These free grouping variables also appear in the grouping predicate. It means that *TG2* depends, as an answer fact to *SQ2*. on all the answer facts to *SQ2* and thus on all the answer facts to *SQG2* as well, and under them on *TG2* itself. Therefore *TG2* depends on itself and there is a cycle in the group dependencies. □

Algorithm 4.1 on reducible aggregate programs in the tail-recursive case has been implemented in the EKS-V1 prototype. This includes the *bill of materials* and the *parts explosion* examples. Experiments on these examples have shown that the cost of controlling the subquery completion represents about 10% of the total evaluation cost.

Example 4.1 (continued)
In the case where the first variable *Part* is instantiated in the query literal, the reduced program is tail-recursive and there is no need to add any reduced literals. During the evaluation of a query *?- part_subpart_qty(*P, SP, Qty)*, a subquery *part_subpart_qty*(**IP*, S_i, Q_i) may depend on itself (actually on a variant of itself) if and only if there is a cycle in the *assembly* relation. These dependencies correspond to several group dependencies with the same value **IP*

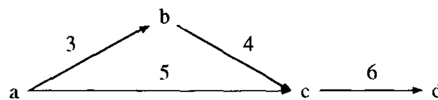

Fig. 3 Data for the assembly relation.

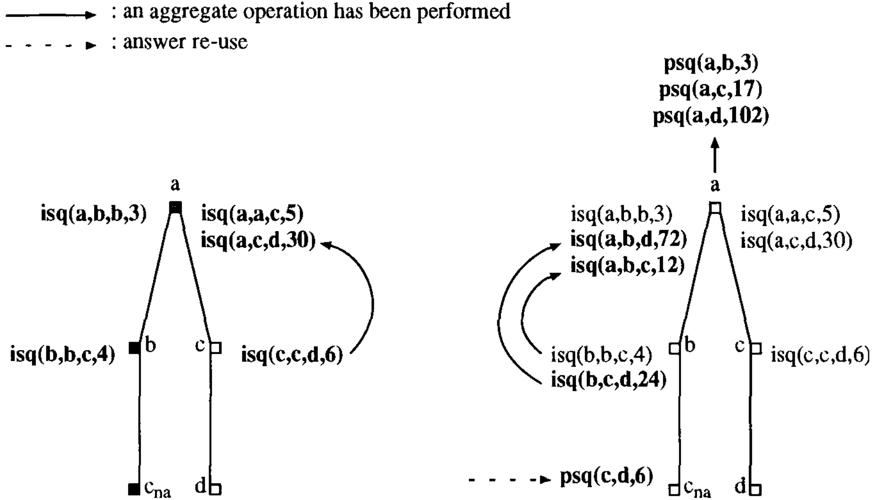

Fig. 4 Subquery completion mechanism on part_subpart_qty.

for the first argument. The evaluation of queries for this pattern under subquery completion is complete and correct in the acyclic case, and fails if the assembly relation is cyclic.

Consider the following facts for the relation *assembly* represented in Fig. 3:

```
assembly(a, b, 3).
assembly(b, c, 4).
assembly(a, c, 5).
assembly(c, d, 6).
```

Suppose that the query is *?- part_subpart_qty*(a, P, Q). Figure 4 represents the subquery tuples on the predicate *part_subpart_qty* during the evaluation. We abbreviate the predicates *part_subpart_qty* and *int_subpart_qty* with *psq* and *isq* respectively. For simplicity reasons, we do not represent the *psq* facts except for the top query and the re-used answers. A completed subquery is represented by a white square. A tuple annotated by "*na*" is non-admissible. At the end of the first stage of the evaluation, subqueries have been derived, among them a non-admissible subquery for c. The original subquery for c is being answered and has not been completed yet. Some partial answering is generated for the subquery on a which cannot be aggregated, because the subquery on a has not been completed yet. Only during the second stage, the answers for c can be re-used for answering the non-admissible subquery. Note that the subquery for c is answered only once. The aggregate operation for the group of a may then be performed, as all the subqueries underneath have been completed. □

§5 Related Work

5.1 Analogy with Negation

We have already pointed out the analogy between aggregates and negation. There is a correspondence between the two notions of stratification in both areas, and between group stratification on the one hand and dynamic stratification[26] (or effective stratification,[3] constructive consistency,[8] or modular stratification[29]) on the other. The coordination issue is essentially the same for negation (stratified case) and for aggregates (stratified case). Indeed, just as for aggregate predicates, negated subqueries must be fully answered before negated facts can really be inferred. The evaluation of stratified negation has been studied in the top-down evaluation framework in Refs. 20) and 34). It has also been addressed in the framework of rewriting methods for bottom-up computation such as Magic Sets[4,6] and Alexander.[18] The issue arises when rewriting a stratified program using the Magic Sets rewriting method: the rewritten program is *not* in general stratified and its non-deterministic bottom-up computation is no longer possible. In order to solve this issue, various labelings of rewritten rules have been introduced, in order to ensure that negated (or aggregate) subqueries are answered fully before their answers can be used. For instance, in Ref. 18) Kerisit proposes to classify the rewritten rules along the strata of the original program. The bottom-up fixpoint normally applied to the rewritten rules is modified in order to take the strata numbers attached to the rules into account: at a given point of time during the evaluation, only the rewritten rules corresponding to the current lowest stratum can be activated. When no more operations can take place on this stratum the rules for the immediately higter stratum are re-activated, etc. This algorithm applies to the stratified aggregate problem in a straightforward manner.

Methods have been proposed in Refs. 3) and 8) for implementing the evaluation of negation in the dynamic stratified case. The Ordered_Search technique[31] has also been used for implemeting the evaluation of modularly stratified programs.

5.2 Other Work on Aggregates

In Ref. 17), Klug first formalized aggregates in relational algebra and calculus, and argued that the notion of duplicates (multi-sets) was not needed for the expressivity of aggregates. We also think that the notion of multi-sets is not necessary for specifying semantics unlike.[23] We regard the problem of being able to handle full duplicates within sets as an issue independent from aggregate computation. It is rather a modeling issue, as to how one may want to represent the data for a given application. Our standpoint however still permits a correct solution to the duplicate issue in the computation of aggregates: it can be performed by choosing the arguments present in the grouping predicate. We give

an example illustrating this point.

Example 5.1 Duplicates in the computation of aggregates
In Example 1.1, the grouping predicate is the whole *employee* relation. Therefore, if several employees have the same salary, the duplicate salaries are considered. However, if one wants to compute the average of the *distinct* salary values by department, one may do so by simply defining the following predicate:

```
avg_distinct_salary_per_dept(Dept, AvgSal) <-
        group_by(
                salary_dept(Dept, Salary),
                [Dept],
                [AvgSal isagg avg(Salary)]
                ).
```

where *salary_dept* is a projection of the *employee* relation leaving out the *name* attribute:

```
salary_dept(Dept, Salary) <-
        employee(_Name, Dept, Salary).
```

□

The model that we consider remains a *flat* model: it does not allow set-valued (or nested) attributes. In other words, sets are not first-class objects in EKS-V1. Hence, we are not following the research trend in nested relations, NF2 models, represented for instance by research projects such as COL[1)] or LDL.[39)] In these approaches, a more general grouping (or nesting) facility is provided allowing aggregate functions to be simply expressed as functions applied to set-valued attributes. We believe that the extension of a flat model with (scalar) aggregate facilities (chosen here as in Refs. 23) and 11)) remains worth investigating because its requirements on the physical level (storage and manipulation) are less stringent, it represents a natural extension of Datalog systems and despite its restrictions, it may well cover an important part of the application requirements.

Our work is close to that on Traversal Recursion[27)] in the way we consider aggregate operations as operations on top of graph traversals. However we generalize graph traversal to more complex structures than graphs and we do not incorporate the semantics of the particular aggregate function (min/max) and thus never allow cyclic graphs. Although this leads to some restrictions, we believe that, if one takes semantics of the aggregate functions into account, this should be done within as formal and as general a framework as possible.

Several recent papers[23,11,19,7,30,41,36)] also consider aggregates in Datalog programs. These papers take a model theoretic approach for defining the semantics of aggregate programs. The work in Refs. 30) and 36) unifies the other approaches in a more general framework. Recently, Van Gelder[42)] has extended the semantics to a wider class of aggregate programs by reconsidering the

underlying semantics of the aggregate operations themselves.

In the *stratified aggregate* case, the semantics and evaluation methods proposed are equivalent to ours. In Ref. 23), Mumick *et al.* extend the Magic transformation producing so-called *magic stratified* programs. The evaluation of such programs can be performed in an order corresponding to the stratification order of the original program by a modification of the bottom-up fixpoint, just like the case of negation.[18]

For defining the *semantics of non-stratified aggregate programs*, the approach taken in Refs. 23), 11), 19), 30), 36) and 42) is different from ours: in these papers, the authors do not separately consider the underlying reduced program. Instead, they take into account the semantics of the aggregate operations, as well as the other arithmetic constructs appearing in an aggregate program, in order to define semantics. This allows them to treat the class of *monotonic aggregate* programs (like the *minimal length path* program or the so-called *corporate takeover* program) for which natural semantics exists. In Ref. 11), Consens and Mendelzon also treat *closed semiring* programs as a special case of recursive aggregate programs having natural semantics.

The *evaluation* of recursive aggregate programs is not addressed in Ref. 23). Mumick *et al.* simply mention that an evaluation following the order of the groups would be possible (which seems to be quite easy to realize). In Ref. 11), Consens and Mendelzon propose a general algorithm applying to closed semiring or to monotonic aggregate programs. Closed semirings are also interesting because specialized algorithms relying on graph traversal (such as in Ref. 12)) can be used for their evaluation. The case of monotonic programs involving minimum and maximum predicates has been the object of other recent papers,[15,16] proposing a bottom-up evaluation mechanism called *greedy fixpoint*, which coincides with Dijkstra's algorithm on the shortest path problem. In Ref. 13), Dietrich presents an algorithm based on extension tables, which applies to the shortest path problem, and is also equivalent to Dijkstra's algorithm on this problem. Surarshan and Ramakrishnan in Ref. 35) optimise the evaluation of programs with extrema by removing irrelevant facts. The *parts explosion* Example 3.2 is also treated in Ref. 25). Phipps uses a procedural language, where the control of the completion for each subquery during query evaluation is expressed in the program by the user.

Recently, a control technique called Ordered_Search[31] has been developed, which works on the transformed programs obtained by Magic Templates rewriting of (left-to-right) modularly stratified or recursive aggregate programs. This technique is very similar to subquery completion, in that it keeps track of the dependencies between the magic facts (i.e. the subqueries) in order to control the evaluation of recursive aggregate programs. Ordered_Search has been implemented in the Coral deductive database system.[32] The data structure used in Ordered_Search has to implement the subquery dependencies, whereas these dependencies are already part of the run-time data structure in EKS-V1.

§6 Conclusion

Our contribution has been to provide an efficient evaluation mechanism for group stratified reducible aggregate programs. An extension to the work presented here would be to extend our solution to the classes of closed semirings and monotonic aggregate programs, which indeed have "natural" semantics. We are also considering an extension of the subquery completion mechanism to the evaluation of modularly stratified programs with negation. As Ordered_Search is also used for the evaluation of such programs, we believe that such an extension of the subquery completion mechanism is feasible. Finally, little work seems to have been done in order to push selections into the computation of aggregates: in the query "*sum_salary_per_dept(D, Sum) and Sum* < *100,000*", the constant 100,000 could be used in order to stop the aggregation, as soon as this constant is overtaken.

Acknowledgements

The author is indebted to Laurent Vieille for many constructive comments on earlier versions of this paper. I have also benefited from comments from several colleagues at ECRC, as well as useful remarks from Eric Villemonte de la Clergerie, Gill Dobbie, Sundararajarao Sudarshan, Raghu Ramakrishnan, and anonymous referees.

References

1) Abiteboul, S. and Grumbach, S., "A Rule-Based Language with Functions and Sets," *ACM Transactions on Database Systems, 16, 1*, pp. 1-30, March 1991.
2) Bidoit, N. and Froidevaux, C., "Negation by Default and Unstratifiable Logic Programs," *Technical Report, 437*, LRI, Orsay, 1988. To appear in a special issue of *TCS on Research in Deductive Databases*.
3) Bidoit, N. and Legay, P., "WELL! An Evaluation Procedure for All Logic Programs," in *Proc. of the 3rd Int. Conference on Database Theory* (*ICDT*), Paris, France, December 1990.
4) Balbin, I., Meenakshi, K., and Ramamohanarao, K., "A Query Independent Method for Magic Set Computation on Stratified Databases," in *Proc. of the International Conference on Fifth Generation Computer Systems* (*FGCS*), Tokyo, Japan, pp. 711-718, November 1988.
5) Beeri, C. and Ramakrishnan, R., "On the Power of Magic," in *Proc. of the 6th ACM Symposium on Principles of Database Systems* (*PODS*), San Diego, California, March 1987.
6) Beeri, C., Ramakrishnan, R., Srivastava, D., and Sudarshan, S., "Magic Implementation of Stratified Logic Programs," *Technical Report*, unpublished manuscript, August 1990.
7) Beeri, C., Ramakrishnan, R., Srivastava, D., and Sudarshan, S., "The Valid Model Semantics for Logic Programs," in *Proc. of the 11th ACM Symposium on Principles of Database Systems* (*PODS*), San Diego, California, June 1992.

8) Bry, F., "Logic Programming as Constructivism: A Formalization and Its Application to Databases," in *Proc. of the 8th ACM Symposium on Principles of Database Systems (PODS)*, Philadelphia, Pennsylvania, pp. 34-50, March 1989.
9) Bry, F., "Query Evaluation in Recursive Databases: Bottom-Up and Top-Down Reconciled," in *Proc. of the 1st International Conference on Deductive and Object-Oriented Databases (DOOD)*, Kyoto, Japan, pp. 95-112, 1989.
10) Brodsky, A. and Sagiv, Y., "On Termination of Datalog Programs," in *Proc. of the 1st International Conference on Deductive and Object-Oriented Databases (DOOD)*, Kyoto, Japan, pp. 95-112, 1989.
11) Consens, M. P. and Mendelzon, A. O., "Low Complexity Aggregation on GraphLog and Datalog," in *Proc. of the 3rd Int. Conference on Database Theory (ICDT)*, Paris, December 1990.
12) Cruz, I. and Norvell, T., "Aggregative Closure: An Extension of Transitive Closure," in *Proc. IEEE 5th International Conference on Data Engineering*, pp. 384-391, February 1989.
13) Dietrich, S. W., "Shortest Path by Approximation in Logic Programs," *ACM Letters on Programming Languages and Systems, 1, 2*, pp. 119-137, June 1992.
14) Freeston, M., "The BANG File: A New Kind of Grid File," in *Proc. of the ACM SIGMOD Conference on Management of Data*, San Francisco, California, pp. 260-269, May 1987.
15) Ganguly, S., Greco, S., and Zaniolo, C., "Minimum and Maximum Predicates in Logic Programming," in *Proc. of the 10th ACM Symposium on Principles of Database Systems (PODS)*, Denver, Colorado, May 1991.
16) Ganguly, S., Greco, S., and Zaniolo, C., "Greedy by Choice," in *Proc. of the 11th ACM Symposium on Principles of Database Systems (PODS)*, San Diego, California, June 1992.
17) Klug, A., "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions," *Journal of the ACM, 29, 3*, pp. 699-717, July 1982.
18) Kerisit, J. M. and Pugin, J. M., "Efficient Query Answering on Stratified Databases," in *Proc. of the International Conference on Fifth Generation Computer Systems (FGCS)*, Tokyo, Japan, pp. 719-725, November 1988.
19) Kemp, D. and Stuckey, P., "Semantics of Logic Programs with Aggregates," in *Proc. of the International Logic Programming Symposium*, 1991.
20) Kemp, D. B. and Topor, R. W., "Completeness of a Top-Down Query Evaluation Procedure for Stratified Databases," in *Proc. of 5th Int. Conference and Symposium on Logic Programming* (R. A. Kowalski and K. A. Bowen, eds.), Seattle, WA, pp. 178-194, August 1988.
21) Lefebvre, A., "Towards an Efficient Evaluation of Recursive Aggregates in Deductive Databases," in *Proc. of the International Conference on Fifth Generation Computer Systems (FGCS)*, Tokyo, Japan, pp. 915-925, June 1992.
22) Lefebvre, A. and Vieille, L., "On Deductive Query Evaluation in the DedGin* System," in *Proc. of the 1st International Conference on Deductive and Object-Oriented Databases (DOOD)*, Kyoto, Japan, pp. 95-112, 1989.
23) Mumick, I. S., Pirahesh, H., and Ramakrishnan, R., "The Magic of Duplicates and Aggregates," in *Proc. of the 16th VLDB Conference*, Brisbane, Australia, pp. 264-277, August 1990.
24) Naughton, J. F., Ramakrishnan, R., Sagiv, Y., and Ullman, J. D., "Efficient Evaluation of Rigth-, Left-, and Multi-Linear Rules," in *Proc. of the ACM SIGMOD Conference on Management of Data*, Portland, Oregon, pp. 235-242, June 1989.
25) Phipps, G., "Glue: A Deductive Database Programming Language," in *Proc. of the*

NALCP'90 Workshop on Deductive Databases (Jan Chomicki, ed.), pp. 1-6, October 1990. Extended Abstract.

26) Przymusinski, T. C., "Every Logic Program Has a Natural Stratification and an Iterated Fixed Point Model," in *Proc. of the 8th ACM Symposium on Principles of Database Systems* (*PODS*), Philadelphia, Pennsylvania, pp. 11-21, March 1989.
27) Rosenthal, A., Heiler, S., Dayal, U., and Manola, F., "Traversal Recursion: A Practical Approach to Supporting Recursive Applications," in *Proc. of the ACM SIGMOD Conference on Management of Data*, Washington D. C., May 1986.
28) Rohmer, J., Lescoeur, R., and Kerisit, J.-M., "The Alexander Method: A Technique for the Processing of Recursive Axioms in Deductive Databases," *New Generation Computing*, *4*, *3*, pp. 273-285, 1986.
29) Ross, K., "Modular Stratification and Magic Sets for DATALOG Programs with Negation," in *Proc. of the 9th ACM Symposium on Principles of Database Systems* (*PODS*), Nashville, Tennessy, pp. 161-171, April 1990.
30) Ross, K. and Sagiv, Y., "Monotonic Aggregation in Deductive Databases," in *Proc. of the 11th ACM Symposium on Principles of Database Systems* (*PODS*), San Diego, California, June 1992. Also presented at the ILPS'91 Workshop on Deductive Databases.
31) Ramakrishnan, R., Srivastava, D., and Sudarshan, S., "Controlling the Search in Bottom-Up Evaluation," in *Proc. of the Joint Int. Conference and Symposium on Logic Programming*, Washington D. C., pp. 273-287, November 1992.
32) Ramakrishnan, R., Srivastava, D., Sudarshan, S., and Seshadri, P., "Implementation of the CORAL Deductive Database System," in *Proc. of the ACM SIGMOD Conference on Management of Data*, Washington D. C., May 1993.
33) Seki, H., "On the Power of Alexander Templates," in *Proc. of the 8th ACM Symposium on Principles of Database Systems* (*PODS*), Philadelphia, Pennsylvania, pp. 150-159, March 1989.
34) Seki, H. and Itoh, H., "A Query Evaluation Method for Stratified Programs under the Extended CWA," in *Proc. of 5th Int. Conference and Symposium on Logic Programming* (R. A. Kowalski and K. A. Bowen, eds.), Seattle, WA, pp. 195-211, August 1988.
35) Sudarshan, S. and Ramakrishnan, R., "Aggregation and Relevance in Deductive Databases," in *Proc. of the 17th VLDB Conference*, Barcelona, Spain, September 1991.
36) Sudarshan, S., Srivastava, D., Ramakrishnan, R., and Beeri, C., "Extending the Well-Founded and Valid Semantics for Aggregation," in *Proc. of the International Logic Programming Symposium*, Vancouver, British Columbia, October 1993.
37) Sacca, D. and Zaniolo, C., "Magic Counting Methods," in *Proc. of the ACM SIGMOD Conference on Management of Data*, San Francisco, California, pp. 49-59, May 1987.
38) Tamaki, H. and Sato, T., "OLD Resolution with Tabulation," in *Proc. of the 3rd Int. Conference on Logic Programming*, London, UK, pp. 84-98, June 1986.
39) Tsur, S. and Zaniolo, C., "LDL: A Logic-Based Data-Language," in *Proc. of the 12th VLDB Conference*, Kyoto, Japan, pp. 33-41, August 1986.
40) Ullman, J. D., "Bottom-Up Beats Top-Down for Datalog," in *Proc. of the 8th ACM Symposium on Principles of Database Systems* (*PODS*), Philadelphia, Pennsylvania, pp. 140-149, March 1989.
41) Van Gelder, A., "The Well-Founded Semantics of Aggregation," in *Proc. of the 11th ACM Symposium on Principles of Database Systems* (*PODS*), San Diego, California, June 1992.
42) Van Gelder, A., "Foundations of Aggregation in Deductive Databases," in *Proc. of the 3rd International Conference on Deductive and Object-Oriented Databases* (*DOOD*),

Phoenix, Arizona, December 1993.

43) Vieille, L., Bayer, P., Küchenhoff, V., and Lefebvre, A., "EKS-V1, A Short Overview," in *AAAI Workshop on Knowledge Base Management Systems* (E. Mays, ed.), Boston, USA, July 1990.

44) Vieille, L., "Recursive Axioms in Deductive Databases: The Query/SubQuery Approach," in *Proc. 1st Int. Conference on Expert Database Systems* (L. Kerschberg, ed.), Charleston, SC, USA, pp. 179-193, April 1986.

45) Vieille, L., "From QSQ towards QoSaQ: Global Optimization of Recursive Queries," in *Proc. 2nd Int. Conference on Expert Database Systems* (L. Kerschberg, ed.), Tysons Corner, Virginia, pp. 421-434, April 1988.

46) Vieille, L., "Recursive Query Processing: The Power of Logic," *Theoretical Computer Science*, *69*, *1*, December 1989.

Dr. Alexandre Lefebvre: He is an engineer at Data and Knowledge Management, at Bull, France. Previously, he was a researcher at the European Computer-Industry Research Centre in Munich, Germany, and later a Research Fellow at Griffith University in Brisbane, Australia. He received a Ph. D. in computer science in 1991 from the University of Paris 5. He is a member of the ACM and the Association for Logic Programming. His professional interests include deductive and object-oriented databases.