PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Towards an Efficient Evaluation of Recursive Aggregates in Deductive Databases

Alexandre Lefebvre[1]

C.I.T., Griffith University
Nathan, QLD 4111, Australia
ajl@cit.gu.edu.au

Abstract

This paper is devoted to the evaluation of aggregates (*avg, sum, ...*) in deductive databases. Aggregates have proved to be a modeling tool necessary 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.

1 Introduction

This paper examines an advanced functionality of deductive database systems, namely the ability to express programs involving both recursion and 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 semantics, evaluation model and implementation of aggregates in the EKS-V1 system [VBKL90].

The recursive aggregate facility is one of the innovative features of the declarative language of EKS-V1, in addition to more standard features like recursion, negation and universal and existential quantifiers. EKS-V1 also provides an extensive integrity checking facility and sophisticated update primitives (hypothetical reasoning, conditional updates). EKS-V1 was developed mainly 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, ...).

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*) computing 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 the general evaluation frameworks such a QSQ or Magic Sets. In the case of EKS-V1, this is performed within the top-down QSQ/DedGin* framework which was developed in [Vie86, Vie88, Vie89] and for which compilation and implementation techniques in a set-oriented way were developed in the DedGin* prototype [LV89]. 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 along a Magic Sets like strategy [RLK86, BR87, SZ87] and a "top-down" strategy [Vie86, Vie88, TS86] (see [Bry89b, Sek89, Ull89, Vie89] 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 are not allowed to be interleaved. In other words, an aggregate value 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.

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

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 computed 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 in a bicycle) and 3) sum up the resulting costs (aggregate computation). Step 1 consists in 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 [RHDM86]. 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 computed 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 fully computed, and another one for which only a partial set of tuples have been produced. 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 evaluation 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 [MPR90, CM90] have been developed in parallel to our work. They take a model-theoretic approach, as we consider a proof-theoretic approach to the semantics of aggregate programs. [MPR90] describes 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.

The structure of this article 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 in section 4 which relies on the notion of subquery completion. Section 5 discusses related work, summarizes the paper and opens towards 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 [MPR90], by an* **aggregate rule** *in the following way:*

agg_pred($\vec{\text{Out}}$) ← **group_by**(
group_pred($\vec{\text{In}}$),
List_of_Grouping_Variables,
List_of_Aggspecs
).

where:

- List_of_Grouping_Variables *is a list of variables.* $\vec{\text{Out}}$ *and* $\vec{\text{In}}$ *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 e.g. 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 [BS89] - 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 soon as the data is acyclic.

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 Aggregate on a base relation

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 *by 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.

Similarly to Datalog programs with stratified negation, a stratified aggregate program P can be divided into *strata* S_i, $i = 1,\ldots,n$.

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 $\bigcup_{j\leq i} S_j$.

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 aggregateed.*

□

Note that this proof-theoretic definition of the semantics is equivalent to the model-theoretic one given in [MPR90, CM90] (see proof in [Lef91]).

2.2 Evaluation

We present here an evaluation algorithm integrated in the QSQ framework. [MPR90] extend the Magic Set formalism to the stratified aggregates in a similar way.

2.2.1 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 by 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.

2.2.2 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. 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.

2.2.3 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. In case of aggregates (also in case of negation), however, *subqueries must be completely answered 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 different from the final count.

In EKS-V1, in order to implement this strategy, we make use of a run-time structure described in [Vie88, LV89] 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 immediately 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. Please refer to [Vie88, LV89] for a precise definition of the DFG. This structure is quite 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 to* N.

3 Semantics of Recursive Aggregates

In order to introduce problems arising in case of recursive aggregate programs, we discuss the classical *bill of materials* example, also presented in [MPR90, CM90].

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 *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 be the semantics of the *bill of materials* example if there were a cycle in the data: which would be the cost of a recursively defined composite part (its value depending 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) 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. But, accepting the first case without accepting the second one would violate this 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 mutually recursive with an aggregate predicate *agg_pred*. We first say that two variables X and Y are *directly connected* if they appear in the same external predicate. Furthermore, consider a predicate *pred* mutually recursive with *agg_pred*. If X appears as the i^{th} argument of *pred* in the head of a rule defining *pred*, and Y appears as the i^{th} argument in a body occurrence of *pred*, then X and Y are also directly connected. The *connected* relationship is finally the transitive closure of the *directly connected* relationship. A variable X in P is said to be **aggregate connected** if X is connected to an aggregate variable or a variable to-be-aggregated.

Obtaining a reduced program from an original program P will be possible if the grouping variables, representing the essence of the program, can be isolated from the aggregate connected variables.

Definition 3.1 Reducible aggregate program

The program P *is said to be* **reducible** *if no grouping variable is aggregate connected. If* P *is reducible, its reduced program* reduce(P) *is obtained by 1) deleting from any rule of* P *any external predicate containing aggregate connected variables and 2) replacing each literal mutually recursive with* agg_pred *by a new predicate 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 carrying the essence of the program.

The full definition of the transformation can be found in [Lef91]. From now on, we consider only reducible aggregate programs[2].

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 having thus 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.
```

The aggregate connected variables in this program are Qty, IQty *and* IQty1. *No grouping variables are aggregate connected 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*

[2] In practice, the only reasonable recursive aggregate programs we could think of are reducible. This is also the case of all examples treated in the related work.

such as I : F ← ..., F', ... *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.* □

We can now define the semantics of a group stratified program P over DB, by refining the definition 2.1. Again, the notion of group stratified programs here is identical to the one proposed in [MPR90].

This time, we note that the facts in *reduce(P)* can be divided along *group strata* GS_i, $i = 1,\ldots,n$, 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 using consecutively 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 aggregate stratified 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 as described in section 2.2.3 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 brought at the data level instead of at the predicate level. [MPR90] remark that group stratified programs can be evaluated in the order of the groups. We give in this section a precise algorithm performing 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 [Vie88]). In section 4.1 we first 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 make subquery dependencies and group dependencies correspond. 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 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.

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, in front of each grouping and each aggregate literal in the body of rules, the corresponding reduced literal. The evaluation of the reduced literal will provide bindings for all the grouping variables. Let P' be the obtained program.*
3. *Modify the query by adding the corresponding reduced literal.*
4. *Evaluate the modified query under subquery completion over reduce(P)$\bigcup P'$.*

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 (see proof in [Lef91]).

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 always exists a non-completed subquery (which depends on itself), and the evaluation stops.

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. However, the group dependencies are cycle free for this example as soon 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.
```

As one can see, 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 anyways. It can be removed.* ∇

4.3 Simplification in the Tail-Recursive Case

The mechanism we have just presented has a main 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 [LV89], and also corresponds to the *right- and left-linear recursive case* as in [NRSU89]. A tail-recursive program is characterized by the following property: for a given subquery, the variables not shared between the head and the body

literal for the recursive predicate are instantiated, and the free variables of the head and the body literal have the same positions in those literals.

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.

Example 4.1 (continued)

In case the first variable Part *is instantiated in the query literal, the 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 *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.* ▽

5 Related Work

[Klu82] has 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 needed for specifying semantics. 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 being present in the grouping predicate. 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 here the research trend around nested relations, NF2 models, represented for instance by research projects such as COL or LDL [AG91, TZ86]. 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 [MPR90, CM90]) 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 by [RHDM86] 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 [MPR90, CM90] [KS91, RS92] also consider aggregates in Datalog programs. These papers take a model theoretic approach for defining the semantics of aggregate programs.

In the *stratified aggregate* case, the semantics and evaluation methods proposed are equivalent to ours. [MPR90] 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.

For defining the *semantics of non-stratified aggregate programs*, the approach taken in [MPR90, CM90] [KS91, RS92] is different from ours: they do not consider separately 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 allow 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. [CM90] 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 by [MPR90]. It is simply mentioned there that an evaluation following the order of the groups would be possible (which seems to be quite easy to realize). [CM90] 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 [CN89]) can be used for their evaluation. The case of monotonic programs involving minimum and maximum predicates has been the object of another recent paper [GGZ91], proposing a bottom-up evaluation mechanism called *greedy fixpoint*. The *parts explosion* example 3.2 is also treated in [Phi90]. They use a procedural language, where the control of the completion for each subquery during query evaluation is expressed in the program by the user.

The more recent work of [KS91, RS92] is concerned with the model-theoretic semantics of aggregate programs, and unifies all the other approaches in a more

general framework.

We have also 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 [Prz90] (or effective stratification [BL90] or constructive consistency [Bry89a]) on the other. The coordination issue is essentially the same for negation (stratified case) as for aggregates (stratified case). Indeed, just as for aggregate predicates, negated subqueries must be fully answered before negated facts can really be inferred.

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. For this class of programs, only bottom-up algorithms have been proposed yet, thus unable to focus on relevant data.

6 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 anonymous referees.

References

[AG91] S. Abiteboul and S. Grumbach. A Rule-Based Language with Functions and Sets. *TODS*, 16(1):1–30, March 1991.

[BL90] N. Bidoit and P. Legay. *WELL!* An Evaluation Procedure for All Logic Programs. In *Proc. of the 3rd Int. Conference on Database Theory (ICDT)*, Paris, France, December 1990.

[BR87] C. Beeri and R. Ramakrishnan. On the Power of Magic. In *Proc. of the 6th ACM Symposium on Principles of Database Systems (PODS)*, San Diego, California, March 1987.

[Bry89a] F. Bry. Logic Programming as Constructivism: A Formalization and its Application to Databases. In *Proc. of the 8th ACM Symposium on Principles of Database Systems (PODS)*, pages 34–50, Philadelphia, Pennsylvania, March 1989.

[Bry89b] F. Bry. 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)*, pages 95–112, Kyoto, Japan, 1989.

[BS89] A. Brodsky and Y. Sagiv. On Termination of Datalog Programs. In *Proc. of the 1st International Conference on Deductive and Object-Oriented Databases (DOOD)*, pages 95–112, Kyoto, Japan, 1989.

[CM90] M. P. Consens and A. O. Mendelzon. Low Complexity Aggregation on GraphLog and Datalog. In *Proc. of the 3rd Int. Conference on Database Theory (ICDT)*, Paris, December 1990.

[CN89] I. Cruz and T. Norvell. Aggregative Closure: An Extension of Transitive Closure. In *Proc. IEEE 5th International Conference on Data Engineering*, pages 384–391, February 1989.

[GGZ91] S. Ganguly, S. Greco, and C. Zaniolo. Minimum and Maximum Predicates in Logic Programming. In *Proc. of the 10th ACM Symposium on Principles of Database Systems (PODS)*, Denver, Colorado, May 1991.

[Klu82] A. Klug. Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions. *Journal of the ACM*, 29(3):699–717, July 1982.

[KS91] D. Kemp and P. Stuckey. Semantics of Logic Programs with Aggregates. In *Proc. of the International Logic Programming Symposium*, 1991.

[Lef91] A. Lefebvre. Recursive Aggregates in the EKS-V1 System. Technical Report TR-KB-34, ECRC, February 1991.

[LV89] A. Lefebvre and L. Vieille. On Deductive Query Evaluation in the DedGin* System. In *Proc. of the 1st International Conference on Deductive and Object-Oriented Databases (DOOD)*, pages 95–112, Kyoto, Japan, 1989.

[MPR90] I. S. Mumick, H. Pirahesh, and R. Ramakrishnan. The Magic of Duplicates and Aggregates. In *Proceedings of the 16th VLDB Conference*, pages 264–277, Brisbane, Australia, August 1990.

[NRSU89] J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman. Efficient Evaluation of Right-, Left-, and Multi-Linear Rules. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 235–242, Portland, Oregon, June 1989.

[Phi90] G. Phipps. Glue: A Deductive Database Programming Language. In Jan Chomicki, editor, *Proc. of the NALCP '90 Workshop on Deductive Databases*, pages 1–6, October 1990. Extended Abstract.

[Prz90] T. C. Przymusinski. Every Logic Program has a natural Stratification and an Iterated Fixed Point

Model. In *Proc. of the 9th ACM Symposium on Principles of Database Systems (PODS)*, pages 11–21, Philadelphia, Pennsylvania, April 1990.

[RHDM86] A. Rosenthal, S. Heiler, U. Dayal, and F. Manola. Traversal Recursion: a Practical Approach to Supporting Recursive Applications. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Washington DC, May 1986.

[RLK86] J. Rohmer, R. Lescoeur, and J.-M. Kerisit. The Alexander Method: a Technique for the Processing of Recursive Axioms in Deductive Databases. *New Generation Computing*, 4(3):273–285, 1986.

[RS92] K. Ross and Y. Sagiv. Monotonic Aggregation in Deductive Databases. In *Proc. of the 11th ACM Symposium on Principles of Database Systems (PODS)*, 1992. Also presented at the ILPS'91 workshop on deductive databases.

[Sek89] H. Seki. On the Power of Alexander Templates. In *Proc. of the 8th ACM Symposium on Principles of Database Systems (PODS)*, pages 150–159, Philadelphia, Pennsylvania, March 1989.

[SZ87] D. Sacca and C. Zaniolo. Magic Counting Methods. In U. Dayal and I. Traiger, editors, *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 49–59, San Francisco, USA, May 1987.

[TS86] H. Tamaki and T. Sato. OLD Resolution with Tabulation. In *Proc. of the 3rd Int. Conference on Logic Programming*, pages 84–98, London, UK, June 1986.

[TZ86] S. Tsur and C. Zaniolo. LDL: A Logic-Based Data-Language. In *Proc. of the 12th VLDB Conference*, pages 33–41, Kyoto, Japan, August 1986.

[Ull89] J. D. Ullman. Bottom-Up Beats Top-Down for Datalog. In *Proc. of the 8th ACM Symposium on Principles of Database Systems (PODS)*, pages 140–149, Philadelphia, Pennsylvania, March 1989.

[VBKL90] L. Vieille, P. Bayer, V. Küchenhoff, and A. Lefebvre. EKS-V1, A Short Overview. In E. Mays, editor, *AAAI Workshop on Knowledge Base Management Systems*, Boston, USA, July 1990.

[Vie86] L. Vieille. Recursive Axioms in Deductive Databases: the Query/SubQuery Approach. In L. Kerschberg, editor, *Proc. 1st Int. Conference on Expert Database Systems*, pages 179–193, Charleston, SC, USA, April 1986.

[Vie88] L. Vieille. From QSQ towards QoSaQ: Global Optimization of Recursive Queries. In L. Kerschberg, editor, *Proc. 2nd Int. Conference on Expert Database Systems*, pages 421–434, Tysons Corner, Virginia, April 1988.

[Vie89] L. Vieille. Recursive Query Processing: the Power of Logic. *Theoretical Computer Science*, 69(1), December 1989.