Reformulating Aggregate Queries Using Views

Abhijeet Mohapatra and **Michael Genesereth**
Stanford University
{abhijeet, genesereth}@stanford.edu

Abstract

We propose an algorithm to reformulate aggregate queries using views in a data integration LAV setting. Our algorithm considers a special case of reformulations where aggregates in the query are expressed as views over aggregates in the view definitions. Although the problem of determining whether two queries are equivalent is undecidable, our algorithm returns an equivalent reformulation if one exists.

1 Introduction

The problem of reformulating queries using views has received considerable attention in the database and logic communities because of its relevance to data integration. In a typical LAV (local-as-view) integration setting (Ullman 1989), the data sources and the targets are described as precomputed views over a master schema (see figure 1). Since the tuples of the master schema are not actually stored, the predicates in the master schema have to be reformulated in terms of the data sources to answer queries over the target schema.

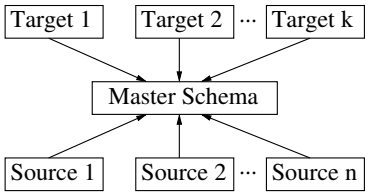


Figure 1: LAV integration setting

The problem of deciding whether an equivalent rewriting of a query can be obtained is undecidable (Duschka and Genesereth 1997). Prior work (Halevy 2001; Afrati and Chirkova 2011; Calvanese et al. 2003; Chirkova, Halevy, and Suciu 2002; Agrawal, Chaudhuri, and Narasayya 2000; Duschka and Genesereth 1997) has proposed techniques to rewrite conjunctive queries using views. For the special case of *conjunctive queries*, the problem of deciding whether an equivalent rewriting exists is NP-Hard (Halevy 2001). The techniques proposed in prior work either return equivalent rewritings for conjunctive queries (Halevy 2001; Afrati and Chirkova 2011; Calvanese et al. 2003; Chirkova, Halevy, and Suciu 2002; Agrawal, Chaudhuri, and Narasayya 2000) or maximally-contained rewritings (Duschka and Genesereth 1997) for other queries when one exists. However, only a small fraction of the prior work on answering queries using views (Cohen, Nutt, and Serebrenik 1999; Calvanese et al. 2003; Afrati, Li, and Ullman 2001; Agrawal, Chaudhuri, and Narasayya 2000; Chirkova, Halevy, and Suciu 2002; Gupta, Harinarayan, and Quass 1995; Srivastava et al. 1996) addresses the case where the query or the views contain aggregates. Prior techniques which reformulate aggregate queries using views consider reformulations using non-recursive views which contain specific aggregates such as min, max, sum and count or *central rewritings* (Afrati and Chirkova 2011) where exactly one view contributes to the aggregate in the head. In contrast, we consider reformulations of aggregate queries using views with *user-defined aggregates* which could potentially have recursive definitions.

In this paper, we consider a special case of reformulations where the aggregates in the query are defined as views over the aggregates in the view definitions. As our underlying query language, we extend Datalog using tuples, sets and aggregates. In section 2, we discuss how user-defined aggregates are specified in our language. Aggregates in our language are defined as predicates over sets. An aggregate could either have a stand-alone definition or be defined as a view over other aggregate predicates. We leverage this modularity to rewrite aggregate queries using views (discussed in section 3.1) when the aggregates in the query and the views are different.

We present an algorithm Invert$_{Agg}$ in section 3.1 which leverages the Inverse Method (Duschka and Genesereth 1997) to reformulate aggregate queries using views. In section 3.2, we establish the following theoretical results about the Invert$_{Agg}$ algorithm.

1. Invert$_{Agg}$ outputs an equivalent reformulation of an aggregate query using views when one such exists.
2. For inductively defined aggregates, the evaluation of a query plan which is generated using Invert$_{Agg}$ is guaranteed to terminate for finite inputs.
3. The size of a query plan which is generated using Invert$_{Agg}$ is linear in the size of the query and the view

definitions.

The Invert$_{Agg}$ algorithm requires aggregates in the query to be modularly expressed as views over the aggregates in the view definitions. However, it might not always be the case that aggregates are modularly defined by a user using other aggregates. In section 4 we show using an example that Invert$_{Agg}$ could potentially be leveraged to reformulate an aggregate with respect to other aggregates even when modular aggregate definitions are absent. We compare our technique to prior work on reformulating aggregate queries using views in section 5.

Before we discuss the problem of reformulating aggregate queries using views and our proposed solution, we motivate the problem through the following scenario.

Running Example: Consider a company XYZ where the employees are paid their monthly salaries using k cheques. For different employees, the respective values of k could potentially be different. We assume that employees are uniquely identified by their name and gender. For the year (say 2012), we record the salary information of an employee in a predicate called salary(Name, Gender, Month, Part, Amount). Suppose a male employee Joe receives the salaries 100 and 120 in the month of January. The corresponding tuples in the predicate salary are salary(joe, m, jan, 1, 100) and salary(joe, m, jan, 2, 120). In addition, we assume that no new employees are either hired or fired from the company in 2012.

For every female employee in XYZ, suppose we would like to compute the average amount that she receives per cheque using a query (say q). We have at our disposal two views v_1(Name, Gender, Salary) and v_2(Name, Gender, Count) which record the annual salary of an employee for 2012 and the number of times an employee receives a cheque in a year. In a LAV integration setting, the master schema consists of the predicate salary. The sources are the views v_1 and v_2 and the target consists of the query q over the master schema. We use this setting (shown in figure 2) in our examples throughout the paper.

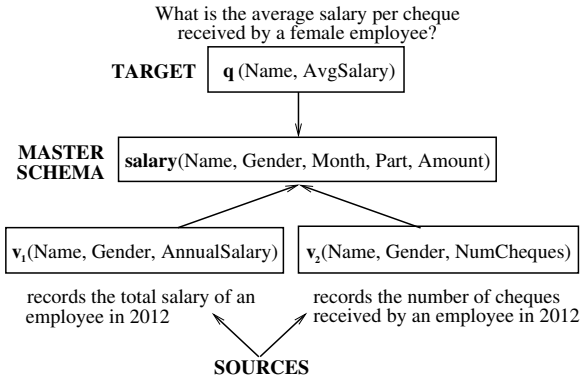


Figure 2: Running Example

Although the aggregates in the query q and the views v_1 and v_2 are different, intuitively we can compute the answer to q using the views v_1 and v_2. Our intuition stems from our knowledge of the average of a set being the ratio of the sum of the set's elements to the cardinality of the set. However in general, relationships between user-defined aggregates are not obvious from their definitions.

2 Background

We extend Datalog using tuples, sets and aggregates to serve as our query language. In our extension we introduce sets as first-class citizens. The elements of a set are either standard datalog atoms or tuples. Examples of sets that are legal in our extension are $\{1, 2, 3\}$, $\{(a, 1), (b, 2)\}$ and $\{(a, 1, 2), (b, 2)\}$. A set could either be empty or be created using the *setof* operator which we define below.

Definition 1. The setof operator has the signature setof($\bar{Y}$, $\phi(\bar{X}, \bar{Y})$, S) and constructs a set $S = \{\bar{Y} \mid \phi(\bar{X}, \bar{Y})\}$ for every binding of values in $\bar{X}$.

We illustrate the construction of sets using the following example.

Example 1. Consider the source $v_1(X, G, S)$ in our running example which records the annual salary S of an employee X in 2012. Suppose we define a view u over v_1 that computes the set of annual salaries per gender as follows.

$$u(G, W) \text{ :- setof}((X, S),\ v_1(X, G, S),\ W)$$

For every value of the variable G in the above definition, the corresponding set W is computed as $\{(X, S) \mid v_1(X, G, S)\}$. Suppose an instance of v_1 contains the set of tuples {(joe, m, 120), (mary, f, 100), (alice, f, 100)}. The corresponding instance of the view u is computed as {(m, {(joe, 120)}), (f, {(mary, 100), (alice, 100)})}.

We note that by leveraging the setof operator we could effectively generate *multisets* under set semantics. For instance, in example 1, the definition of $u(G, W)$ effectively generates a multiset of salaries S for employees of a particular gender.

Since sets are first-class citizens in our language, setof subgoals could potentially generate *nested* sets and tuples as shown in the instance of the view u in example 1. To manipulate sets, we define two operators member() and '|'. The member() operator has the signature member(X, S) where X is a standard datalog atom or a tuple and S is a set. If $X \in S$ then member(X, S) is true otherwise false. The '|' operator is used in our language to represent the decompositions of a set. A set S in our language is decomposed into an element $X \in S$ and the subset $S_1 = S - \{X\}$ as $\{X \mid S_1\}$. For example the decomposition of the set $\{1, 2, 3\}$ into 3 and the subset $\{1, 2\}$ is represented as $\{3 \mid \{1, 2\}\}$.

Defining Aggregates: We define aggregates as predicates over sets. In our language an aggregate could either be defined (a) in a stand-alone manner using the member() and '|' operators or (b) modularly as views over other aggregates. We provide an example below to show how an aggregate could be defined in a stand-alone manner in our language.

Example 2. The following Datalog rules inductively compute the sum A of the i^{th} component of a set of tuples S.

$$\text{sum}(\{\}, 0, I)$$
$$\text{sum}(\{X \mid S\}, A, I) \text{ :- sum}(S, A_1, I),\ A = A_1 + X[I]$$

The first rule specifies the base case of the induction i.e. the sum of an empty set is 0. The second rule inductively defines the sum of the set $S \cup \{X\}$ using the sum of the set S. In the second rule, $X[I]$ represents the i^{th} component of the tuple X. In our running example we could leverage the inductive definition of the sum predicate and the source v_1 to compute the annual salary of all female employees as follows.

$$q_1(A) \text{ :- setof}((X, S),\ v_1(X, \text{f}, S),\ W),\ \text{sum}(W, A, 2)$$

In addition, we could also define an aggregate such as *average* modularly by leveraging other aggregates such as sum and count as follows.

$$\text{average}(W, A, I) \text{ :- sum}(W, S, I), \text{count}(W, C, I), A = \frac{S}{C}$$

The treatment of aggregates as predicates in our language has two advantages. First, we can build complex aggregates modularly using simpler aggregates leading to compact programs. Second, we could potentially leverage this modularity to reformulate aggregate queries using views where the aggregates in the query and the views are different (sections 3.1 and 4).

Flattening Nested Sets and Tuples: Since sets are first-class citizens in our language, nested sets and tuples could potentially be generated using the setof operator. In general, nested sets which have the same flattened representation are *not* equivalent. However for the reformulations which we generate in section 3.1, nested sets with the same flattened representation satisfy a notion of equivalence which is captured using the predicate *equiv* as follows. Two sets S and T are *equivalent* with respect to aggregation i.e. equiv(S, T) is true *iff* S and T have the same flattened representation. We formally capture the flattening of sets using the *expand* function defined below.

Definition 2. Suppose $X = X_1 \circ X_2 \ldots \circ X_k$ and $X \in S$. The function expand(S) contains a tuple Y such that $Y = Y_1 \circ Y_2 \ldots \circ Y_k$ where:

$$\begin{array}{ll} Y_i = X_i & \text{if } X_i \text{ is a tuple,} \\ Y_i \in \text{expand}(X_i) & \text{if } X_i \text{ is a set, and} \\ Y_i = (X_i) & \text{otherwise} \end{array} .$$

3 Reformulating Aggregate Queries

In the previous section we discussed the specification of aggregates in our language. An aggregate query is specified by (a) defining sets using setof operator and (b) manipulating the sets using aggregation predicates. To reformulate an aggregate query using views, we leverage the Inverse Method (Duschka and Genesereth 1997). The basic principle of the Inverse Method is as follows. Consider a Datalog query Q and a set of view definitions V over a common schema. To answer Q using the views in V, the view definitions are *inverted*. An inverse rule is generated for every subgoal in the view definition. The inverse rule contains the view in the body of the rule. The head variables which are not in the view are replaced by skolem functions over the variables in the view. For instance, consider two predicates $r(A, B)$ and $s(B, C)$ and the following view which computes the *join* of the two predicates.

$$v(A, C) \text{ :- } r(A, B), s(B, C)$$

The inverse rules corresponding to the above view definition are as follows.

$$r(A, f(A, C)) \text{ :- } v(A, C)$$
$$s(f(A, C), C) \text{ :- } v(A, C)$$

In the Inverse Method (Duschka and Genesereth 1997) the set of the inverted view definitions V^{-1} and the query Q form a query plan for reformulating a query using views.

In our proposal, we reformulate aggregate queries using views by inverting the view definitions which contain sets and aggregate predicates. As a first step we discuss the inversion of views which contain setof subgoals.

Inverting Views Containing Sets: As discussed in section 2, we construct sets in our language using the setof operator. Consider a generic rule in our language which contains a setof subgoal.

$$v(\bar{X}, W) \text{ :- setof}(\bar{Y},\ \phi(\bar{X}, \bar{Y}, \bar{Z}),\ W)$$

Suppose $\phi(\bar{X}, \bar{Y}, \bar{Z})$ is a conjunction of subgoals S_1, S_2 ... and S_k. To reformulate $\phi(\bar{X}, \bar{Y}, \bar{Z})$ using the view v we *invert* the definition of the view v. We generate k inverse rules, one for every subgoal S_i. The body of the inverse rule is the conjunction of two subgoals: $v(\bar{X}, W)$ and member$(\bar{Y}, W)$. The head of the rule is S_i. The variables of S_i which do not appear in the body are replaced by skolem functions of the variables W and $\bar{X}$. For example, consider the definition of the view u presented in example 1.

$$u(G, W) \text{ :- setof}((X, S),\ v_1(X, G, S),\ W)$$

Suppose we are given an instance of $u(G, W)$. Using the supplied instance and the definition of u, we can derive the tuples of the source v_1 using the following *inverse* rule.

$$v_1(X, G, S) \text{ :- } u(G, W),\ \text{member}((X, S), W)$$

To see why the above rule works, let us revisit the semantics of the setof operator. The view u consists of tuples $u(G, W)$ where W is the set of values $\{(X, S) \mid v_1(X, G, S)\}$. Hence for every tuple $u(G, W)$ there exists a set of $v_1(X, G, S)$ tuples such that $W = \{(X, S) \mid v_1(X, G, S)\}$. Equivalently, member$((X, S), W)$ is true.

However, it is not always the case that we can compute an instance of a subgoal by inverting the view definition. For example consider a view u' which is defined as follows.

$$u'(G, W) \text{ :- setof}(S,\ v_1(X, G, S),\ W)$$

By projecting out the variable X in the definition of the view u' we *lose information* on how to reformulate v_1 tuples using the view u'. Suppose we invert the definition of u' as follows.

$$v_1(f(G, W), G, S) \text{ :- } u'(G, W),\ \text{member}(S, W)$$

Consider an instance of u' containing the tuple (g, w). By applying the inverse rule on (g, w) we can only deduce that there exists a value of X (= $f(g, w)$) such that $v_1(X, g, w)$ is true. However we cannot compute the instance of v_1 which corresponds to a supplied instance of u'.

We now present our algorithm to reformulate aggregate queries using views.

3.1 Reformulation Algorithm

In our query language, we represent aggregates as predicates over sets. The representation allows for complex aggregates (such as average) to be modularly expressed as views over other aggregates (such as sum and count). Suppose an aggregate query contains an aggregate which is defined as a view over the aggregates in the views. In such a case we expand the aggregate in the query. Consider the scenario described in our running example from section 1. The query $q(X, Y)$ which computes the average salary Y per cheque of a female employee X is defined as follows.

$$q(X, Y) \text{ :- setof}((M, P, A), \text{ salary}(X, \text{f}, M, P, A), W), \\ \text{average}(W, Y, 3)$$

Suppose the predicate average is defined as a view over the aggregates sum and count as shown in section 2. Then we expand the definition of average in the query $q(X, Y)$ as follows.

$$q(X, Y) \text{ :- setof}((M, P, A), \text{ salary}(X, \text{f}, M, P, A), W), \\ \text{sum}(W, S, 3), \text{ count}(W, C, 3), Y = \frac{S}{C}$$

In the above expansion, we do not further expand the recursive aggregate definitions (of sum and count).

Consider a generic aggregation query in our language.

$$q'(\bar{X}, A) \text{ :- setof}(\bar{Y}, \phi(\bar{X}, \bar{Y}, \bar{Z}), W), \text{ agg}(W, A), \\ \psi(\bar{X}, \bar{V})$$

The set W in the query q' could potentially be nested. Using the notion of *equivalence* which we define in section 2, if expand(W') = expand(W) for any set W', then the aggregate 'agg' computes the value A on W' as well. Suppose that in our reformulations, two nested sets W and W' are constructed by applying the setof operator on a conjunction of subgoals $\phi(\bar{X}, \bar{Y}, \bar{Z})$. The sets W and W' satisfy the following properties.

Property 1. If $W = W'$, then equiv(W, W') is true.

Property 2. If $W \neq W'$, then

$$\text{setof}(\bar{Z} \circ \bar{Y}, \phi(\bar{X}, \bar{Y}, \bar{Z}) \,\&\, \text{member}(\bar{Y}, W_1), W) \wedge \\ \text{setof}(\bar{Z} \circ (W_1), \phi(\bar{X}, \bar{Y}, \bar{Z}) \,\&\, \text{member}(\bar{Y}, W_1), W') \\ \Leftrightarrow \text{ equiv}(W, W')$$

Using the above properties, we account for all possible nestings of expand(W) in the query $q'(\bar{X}, A)$ by (a) re-labeling the set variables in aggregation predicates as distinct variables and (b) adding equiv(W, W') predicates, where W is a set constructed using a setof subgoal and W' is a re-labeling of W inside an aggregate.

$$q'(\bar{X}, A) \text{ :- setof}(\bar{Y}, \phi(\bar{X}, \bar{Y}, \bar{Z}), W), \text{ agg}(W', A), \\ \psi(\bar{X}, \bar{V}), \text{ equiv}(W, W')$$

Inverting View Definitions: In our proposal, we leverage the Inverse Method (Duschka and Genesereth 1997) to reformulate an aggregate query using views. We generate a query plan by inverting the setof subgoals and the aggregation predicates in the view definitions.

To separate the inversion of setof subgoals and aggregates we introduce *auxiliary predicates* in the views. Consider a generic view definition in our language with a single aggregate *agg*.

$$v(\bar{X}, A) \text{ :- setof}(\bar{Y}, \phi(\bar{X}, \bar{Y}, \bar{Z}), W), \text{ agg}(W, A), \\ \psi(\bar{X}, \bar{V})$$

We replace the setof subgoal in the view definition by an auxiliary predicate (say t) using the following rules.

$$v(\text{X}, A) \quad \text{:- } t(\bar{X}, \bar{Z}, W), \text{ agg}(W, A), \ \psi(X, V)$$
$$t(\bar{X}, \bar{Z}, W) \text{ :- setof}(\bar{Y}, \phi(\bar{X}, \bar{Y}, \bar{Z}), W)$$

There is an implicit functional dependency in an auxiliary predicate from the non-set variables to the set variables. However this dependency is not captured by the setof operator. We capture this dependency using the constraint: $\text{t}(\bar{X}, \bar{Z}, W_1) \wedge t(\bar{X}, \bar{Z}, W_2) \implies W_1 = W_2$. By replacing the setof subgoals in the views using auxiliary predicates we generate a query plan in a two-step process. First we invert the modified view definitions over auxiliary predicates and aggregates. Subsequently we invert the auxiliary predicates.

Input: an aggregate query q and a set of views V
Step 1:
for all aggregates a in q **do**
  if a has a non-recursive definition using aggregates $a_1, a_2 \ldots a_i$ **then**
    Expand a in q as a_i s.
  end if
end for
$q' \leftarrow q$ with expanded aggregate definitions.
Step 2:
Re-label set variables in aggregates to make them distinct. For every set S constructed using a setof subgoal and a relabeling T of S, add equiv(S, T) to the modified query q'.
Step 3:
Replace setof($\bar{Y}, \phi(\bar{X}, \bar{Y}, \bar{Z}), W$) in a view using an **auxiliary predicate** say $t(\bar{X}, \bar{Z}, W)$ which is defined as follows.

$$t(\bar{X}, \bar{Z}, W) \text{ :- setof}(\bar{Y}, \phi(\bar{X}, \bar{Y}, \bar{Z}), W)$$

$T \leftarrow$ set of auxiliary predicate definitions
$V^{-1} \leftarrow$ **inverse** of rules in $V \cup T$
Step 4:
For every auxiliary view $t(\bar{X}, W)$ and aggregation predicate $a(\bar{X}, \bar{A})$, we generate the following **functional dependencies**.

$$t(\bar{X}, W_1) \wedge t(\bar{X}, W_2) \implies W_1 = W_2$$
$$a(\bar{X}, A_1) \wedge a(\bar{X}, A_2) \implies A_1 = A_2$$

where W and $\bar{A}$ are set and aggregate result respectively.
$\Lambda \leftarrow$ set of functional dependencies
Output: $\{q'\} \cup V^{-1} \cup \Lambda$

Figure 3: Algorithm **Invert**$_{Agg}$ to reformulate aggregate queries using views

We present an algorithm Invert$_{Agg}$ to reformulate aggre-

gate queries using views in figure 3. We describe the working of the Invert$_{Agg}$ algorithm using the following example.

Example 3. Consider the query $q(X, Y)$ presented in our running example of section 1 which computes the average amount received per cheque Y by a female employee X.

$$q(X, Y) \text{ :- setof}((M, P, A),\ \text{salary}(X, \text{f}, M, P, A),\ W),\ \text{average}(W, Y, 3)$$

We have two sources $v_1(X, G, S)$ and $v_2(X, G, C)$ which are expressed as views over the predicate salary in the master schema. The view $v_1(X, G, S)$ records the annual salary S of an employee X in the company and is defined as follows.

$$v_1(X, G, S) \text{ :- setof}((M, P, A), \text{salary}(X, G, M, P, A), W),\ \text{sum}(W, S, 3)$$

The view $v_2(X, G, C)$ records the number of cheques C received by an employee X in the year 2012 and is defined as follows.

$$v_2(X, G, C) \text{ :- setof}((M, P, A), \text{salary}(X, G, M, P, A), W),\ \text{count}(W, C, 3)$$

Suppose the predicate average is defined modularly as a view over the aggregates sum and count as in section 2. Using the Invert$_{Agg}$ algorithm we first expand the average predicate in query $q(X, Y)$ as follows.

$$q(X, Y) \text{ :- setof}((M, P, A),\ \text{salary}(X, \text{f}, M, P, A),\ W),\ \text{sum}(W, S, 3),\ \text{count}(W, C, 3),\ Y = \frac{S}{C}$$

We, then, modify the above query to account for all possible nestings of expand(W).

$$q(X, Y) \text{ :- setof}((M, P, A),\ \text{salary}(X, \text{f}, M, P, A),\ W),\ \text{sum}(U, S, 3),\ \text{count}(V, C, 3),\ Y = \frac{S}{C},\ \text{equiv}(W, U),\ \text{equiv}(W, U)$$

Subsequently, we replace the setof subgoals in the sources v_1 and v_2 using an auxiliary predicate (say t).

$$v_1(X, G, S) \text{ :- } t(X, G, W),\ \text{sum}(W, S, 3)$$
$$v_2(X, G, C) \text{ :- } t(X, G, W),\ \text{count}(W, C, 3)$$
$$t(X, G, W) \text{ :- setof}((M, P, A), \text{salary}(X, G, M, P, A), W)$$

After replacing the setof subgoals in the view definitions with the auxiliary predicate t, we generate the following inverse rules V^{-1} for the definitions of v_1, v_2 and t.

$$\begin{aligned}
t(X, G, f(X, G, S)) &\text{ :- } v_1(X, G, S)\\
\text{sum}(f(X, G, S), S, 3) &\text{ :- } v_1(X, G, S)\\
t(X, G, g(X, G, C)) &\text{ :- } v_2(X, G, C)\\
\text{count}(g(X, G, C), C, 3) &\text{ :- } v_2(X, G, C)\\
\text{salary}(X, G, M, P, A) &\text{ :- } t(X, G, W),\ \text{member}((M, P, A),\ W)
\end{aligned}$$

In the definition of the auxiliary view $t(X, G, W)$, the set W is functionally dependent on the variables X and G. Hence we generate the following functional dependency (Λ).

$$\Lambda: \ t(X, G, W_1) \wedge t(X, G, W_2) \implies W_1 = W_2$$

The expansion of the modified query using the inverse rules V^{-1} and the functional dependency Λ is shown in figure 4. Consider step 4 of the expansion. By the definition of setof operator, the set W which is computed by the subgoal setof$((M, P, A),\ \text{member}((M, P, A), W_1),\ W)$ = $\{(M, P, A) \mid \text{member}((M, P, A), W_1)\}$ = $\{B \mid B \in W_1\}$ = W_1. In the step 6 of the expansion, $v_1(X, \text{f}, S)$ and $v_2(X, \text{f}, C)$ are true. Hence the functional dependency Λ is applied to obtain the equality: $f(X, \text{f}, S)$ = $g(X, \text{f}, C)$. Therefore the derivation equiv$(W, f(X, \text{f}, S)) \wedge$ equiv$(W,$ $g(X, \text{f}, C))$ is valid and the query $q(X, Y)$ is reformulated using the plan $\{q\} \cup V^{-1} \cup \Lambda$ as follows.

$$q(X, Y) \text{ :- } v_1(X, \text{f}, S),\ v_2(X, \text{f}, C),\ Y = \frac{S}{C}$$

We note that we did not invert the definition of the aggregates sum and count in example 3. This is because the aggregate in the query (i.e. average) is defined as a view over the aggregates in the views (i.e. sum and count). However when such modular aggregate definitions are absent, it is necessary to invert the definitions of the aggregates in the views. We address such scenarios in section 4.

3.2 Theoretical Results

In section 3.1, we presented the Invert$_{Agg}$ algorithm to reformulate aggregate queries using views. Invert$_{Agg}$ computes equivalent reformulations of a special case of aggregate queries where the aggregates in the query are expressed as views over the aggregates in the views. In this section we present the theoretical results and the properties of the Invert$_{Agg}$ algorithm. First we establish that Invert$_{Agg}$ computes equivalent reformulations of a query containing *setof* subgoals using views. We then leverage this result to establish the soundness of the plans that are generated by Invert$_{Agg}$ to reformulate aggregate queries using views.

Lemma 1. *Invert$_{Agg}$ outputs an equivalent reformulation of a query containing setof subgoals when one exists.*

Proof. Suppose a query q contains k setof subgoals s_1, $s_2, \ldots s_k$. Let us assume that there exists an equivalent reformulation of query q using n views $\{v_i\}$. We show that Invert$_{Agg}$ computes such an equivalent reformulation.

We replace every setof subgoal s_i in the query q using an auxiliary predicate q_i where q_i is defined as q_i :- s_i. In a similar manner, we replace a setof subgoal s_{ij} in v_i using an auxiliary view v_{ij}. The modified query q and the view definition of v contain no setof subgoals. If we show that Invert$_{Agg}$ generates an equivalent reformulation of every auxiliary predicate q_i using the set of auxiliary view definitions $\{v_{ij}\}$ then we can leverage the result presented in (Duschka and Genesereth 1997) to generate an equivalent reformulation of q by inverting the modified view definitions of $\{v_i\}$.

Consider the definition of an auxiliary predicate $q_i(\bar{X}, W)$:- setof$(\bar{Y},\ \phi_i(\bar{X}, \bar{Y}),\ W)$. From our assumption, there exists an equivalent reformulation of q using the set of views $\{v_i\}$. Hence it follows that an equivalent reformulation of q_i also exists using the set of auxiliary views $\{v_{ij}\}$.

1. $q(X, Y)$
2. setof$((M, P, A),\ \text{salary}(X, \text{f}, M, P, A),\ W),\ \text{sum}(U, S, 3),\ \text{count}(V, C, 3),\ Y = \frac{S}{C},\ \text{equiv}(W, U),\ \text{equiv}(W, V)$
3. setof$((M, P, A),\ t(X, \text{f}, W_1)\ \&\ \text{member}((M, P, A), W_1),\ W),\ \text{sum}(U, S, 3),\ \text{count}(V, C, 3),\ Y = \frac{S}{C},\ \text{equiv}(W, U),\ \text{equiv}(W, V)$
4. $t(X, \text{f}, W_1),\ \text{setof}((M, P, A),\ \text{member}((M, P, A), W_1),\ W),\ \text{sum}(U, S, 3),\ \text{count}(V, C, 3),\ Y = \frac{S}{C},\ \text{equiv}(W, U),\ \text{equiv}(W, V)$
5. $t(X, \text{f}, W_1),\ \text{sum}(U, S, 3),\ \text{count}(V, C, 3),\ Y = \frac{S}{C},\ \text{equiv}(W, U),\ \text{equiv}(W, V)$
6. $v_1(X, \text{f}, S),\ v_1(X, \text{f}, S),\ v_2(X, \text{f}, C),\ Y = \frac{S}{C},\ \text{equiv}(W, f(X, \text{f}, S)),\ \text{equiv}(W, g(X, \text{f}, C))$
7. $v_1(X, \text{f}, S),\ v_2(X, \text{f}, C),\ Y = \frac{S}{C},\ \text{equiv}(f(X, \text{f}, S), f(X, \text{f}, S)),\ \text{equiv}(f(X, \text{f}, S), g(X, \text{f}, C))$
8. $v_1(X, \text{f}, S),\ v_2(X, \text{f}, C),\ Y = \frac{S}{C}$

Figure 4: Expansion of $q(X, Y)$ with respect to the query plan in example 3

There is an equivalent reformulation of ϕ_i using q_i. To see why this is the case, consider a tuple $(\bar{x}, w) \in q_i$. By the definition of the setof operator $w = \{\bar{y} \mid \phi_i(\bar{x}, \bar{y})\}$. Hence a tuple $s = (\bar{x}, \bar{y}) \in \phi_i$ *iff* $\exists w$ s.t member$(\bar{y}, w)$ and $q_i(\bar{x}, w)$ are true. Similarly consider an auxiliary view definition $v_{ij}(\bar{X}, W)$:- setof$(\bar{Y},\ \psi_{ij}(\bar{X}, \bar{Y}),\ W)$. There is an equivalent reformulation of $\psi_{ij}(\bar{X}, \bar{Y})$ using $v_{ij}(\bar{X}, W)$ and member$(\bar{Y}, W)$. Invert$_{Agg}$ generates such a reformulation by inverting the definition of the auxiliary view (Step 3 in figure 3). If Invert$_{Agg}$ generates an equivalent reformulation of ϕ_i using the set of subgoals $\{\psi_{ij}\}$, then we are done. Consider the query $q''(\bar{X}, \bar{Y})$:- $\phi_i(\bar{X}, \bar{Y})$ and the set of views $\{v'_{ij}\}$ where v'_{ij} is defined as $v'_{ij}(\bar{X}, \bar{Y})$:- $\psi_{ij}(\bar{X}, \bar{Y})$. Since v'_{ij} is a conjunctive query, we can obtain an equivalent reformulation of q'' using the set of views $\{v'_{ij}\}$ by inverting the view definitions (Duschka and Genesereth 1997). □

We leverage lemma 1 to establish the soundness of the plans that are generated by the Invert$_{Agg}$ algorithm.

Theorem 1. *Invert$_{Agg}$ computes an equivalent reformulation of an aggregate query using views when one exists.*

Proof. Suppose we have an aggregate query q and a set of views $\{v_i\}$. We replace the setof subgoals in the query and the views using auxiliary predicates. The aggregates in the query (say $\{aq_i\}$) are defined modularly as views over the aggregates (say $\{av_i\}$) in the view definitions. In addition every v_i is a non-recursive view over the auxiliary views and the aggregates $\{av_i\}$. Therefore by leveraging lemma 1 and the equivalence result presented in (Duschka and Genesereth 1997), we prove that the plans that are generated by Invert$_{Agg}$ equivalently reformulate aggregate queries using views. □

In our language aggregates can be defined using recursive rules (as shown in section 2). Prior work (Duschka and Genesereth 1997) shows that inverting recursive rules in general could potentially yield programs whose evaluation may not terminate. We show that for a special case of aggregates, the evaluation of the query plans generated by the Invert$_{Agg}$ algorithm is guaranteed to terminate.

Theorem 2. *For inductively defined aggregates, the evaluation of a query plan which is generated using Invert$_{Agg}$ is guaranteed to terminate for finite inputs .*

Proof. Query plans which are generated by Invert$_{Agg}$ may potentially invert recursive aggregate definitions in the views. In such cases, the evaluation of the plan may not terminate. However, in inductive aggregate definitions, an aggregate on a set S is defined in terms of the aggregate on subsets of S. Such a rule when inverted maps the aggregate on a set to the aggregate on its super-set. For example consider the predicate sum(W, S) which is defined in example 2.

$$\text{sum}(\{X|W\}, S, I) \text{ :- } \text{sum}(W, S_1, I),\ S = S_1 + X[I]$$

The inverse of the above rule sum$(W, f(X, W, S), I)$:- sum$(\{X|W\}, S, I)$ maps the sum of the set W to the sum of $\{X\} \cup W$. Since skolem functions appear only in the head of inverse rules, the number of function symbols generated in the evaluation of the inverse rules of an aggregate (such as sum(W, S, I)) is proportional to the size of the power-set of W and hence finite. □

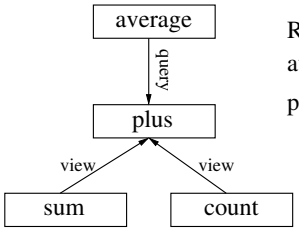
Using theorem 2, we show that the evaluation of query plans which are generated by the Invert$_{Agg}$ algorithm terminates for inductively defined aggregates. We note that although the evaluation of a plan which is generated by Invert$_{Agg}$ could be exponential in the size of the supplied view instances, the size of the plan itself is linear in the size of the query and the view definitions.

Theorem 3. *Invert$_{Agg}$ outputs a query plan which is linear in the size of the query and the view definitions.*

Proof. Consider a query q which is supplied as an input to the Invert$_{Agg}$ algorithm. In the query q, suppose S is a set which is constructed using a setof subgoal and manipulated by an aggregate (say *agg*). Invert$_{Agg}$ relabels the set S as the variable T (say) and adds equiv(S, T) to the query. The number of equiv(S, T) predicates in the modified query is proportional to the number of aggregate predicates in the query. In a subsequent step, Invert$_{Agg}$ rewrites view definitions by replacing every setof subgoal in a view with an auxiliary predicate. The number of auxiliary predicates in a view is, therefore, proportional to the size of the view definition. In the last step, Invert$_{Agg}$ generates an inverse rule for every subgoal in a modified view definition and for every predicate within the setof subgoal in an auxiliary predicate. Hence the size of the inverse rules is proportional to the size of the modified view definitions and the auxiliary predicates. The query plan consists of the query and the inverse rules and is thus linear in the size of the query and view definitions. □

4 Reformulating Aggregate Definitions

In our proposal in section 3.1, we modularly express aggregates as views over other aggregates and leverage this modularity to reformulate aggregate queries using views using the Invert$_{Agg}$ algorithm. We now show using an example that we can potentially leverage Invert$_{Agg}$ to reformulate aggregate queries using views even when the aggregates in the query are not expressed as views over the aggregates in the view definitions. We achieve such reformulations by inverting the definitions of the aggregate subgoals in the view which could potentially be recursive.



Reformulation of average
average = query(plus)
plus = view^{-1}(sum, count)

Figure 5: Reformulating average using sum and count

Example 4. In example 3 we define the average of a set modularly using the sum and the count predicates. Let us consider an alternative Datalog program which computes the average A of a set of numbers W by incrementally computing the average of the subsets of W.

$$\text{average}(\{\},\ 0,\ 0,\ \frac{0}{0})$$

$$\text{average}(\{X \mid W\},\ S,\ C,\ A) \text{ :- } \text{average}(W,\ S_1,\ C_1,\ A_1),\ \text{plus}(S_1, X, S),\ \text{plus}(C_1, 1, C),\ A = \frac{S}{C}$$

Suppose we have two aggregates sum and count over a set of numbers which are defined as follows.

$$\text{sum}(\{\},\ 0)$$
$$\text{sum}(X \mid W,\ S) \text{ :- } \text{sum}(W,\ S_1),\ \text{plus}(S_1,\ X,\ S)$$
$$\text{count}(\{\},\ 0)$$
$$\text{count}(X \mid W,\ C) \text{ :- } \text{count}(W, C_1),\ \text{plus}(C_1,\ 1,\ C)$$

It is not obvious that average(W, S, C, A) as defined above could be reformulated using sum(W, S) and count(W, C). In this example, we denote the addition of numbers using a predicate *plus* instead of the operator '+' as in sections 2 and 3. We make this change to specify the aggregates sum, count and average as views over the plus predicate (shown in figure 5). This reduces the problem of reformulating the average predicate using sum and count predicates to the problem of reformulating the plus predicate by inverting the view definitions of sum and count. Suppose we invert the definitions of the aggregates sum and count.

$$\text{sum}(W,\ f(X,W,S)) \text{ :- } \text{sum}(\{X \mid W\}, S)$$
$$\text{plus}(f(X,W,S),\ X,\ S) \text{ :- } \text{sum}(\{X \mid W\}, S)$$
$$\text{count}(W,\ g(X,W,C)) \text{ :- } \text{count}(\{X \mid W\}, C)$$
$$\text{plus}(g(X,W,C),\ 1,\ C) \text{ :- } \text{count}(\{X \mid W\}, C)$$

We claim that by leveraging the above inverse rules we can reformulate the aggregate average(W, S, C, A) using sum(W, S) and count(W, C). Suppose we have a set of numbers W = {4, 2} as our input. The tuples of sum and count corresponding to the input are $\{(\{\}, 0), (\{2\}, 2), (\{4\}, 4), (\{2, 4\}, 6)\}$ and $\{(\{\}, 0), (\{2\}, 1), (\{4\}, 1), (\{2, 4\}, 2)\}$ respectively. By applying the inverse rules to the supplied instances of the sum and the count predicates we generate additional tuples (which contain functional symbols) for the predicates sum, count and plus. A subset of these tuples is presented in tables 1 and 2 respectively. By definition, aggregates are functions from a set to a value. Hence there is a functional dependency from the input set to the computed aggregate value on the set. If step 4 of the Invert$_{Agg}$ algorithm is applied to the aggregates sum and count, then the following dependencies are derived.

Table 1: Tuples of sum and count computed using inverse rules

sum(W, S)	count(W, C)
$(\{2\}, f(4, \{2\}, 6))$	$(\{2\}, g(4, \{2\}, 2))$
$(\{4\}, f(2, \{4\}, 6))$	$(\{4\}, g(2, \{4\}, 2))$
$(\{\}, f(2, \{\}, 2))$	$(\{2\}, g(2, \{\}, 1))$
$(\{\}, f(4, \{\}, 4))$	$(\{4\}, g(4, \{\}, 1))$

Table 2: Tuples of plus computed using Table 1 and inverse rules

plus(X, Y, Z)
$(f(4, \{\}, 4), 4, f(2, \{4\}, 6))$
$(g(4, \{\}, 1), 1, g(2, \{4\}, 2))$
$(f(, \{\}, 2), 2, f(4, \{2\}, 6))$
$(g(2, \{\}, 1), 1, gf(4, \{2\}, 2))$

$$\text{sum}(W, S_1) \wedge \text{sum}(W, S_2) \implies S_1 = S_2$$
$$\text{count}(W, S_2) \wedge \text{count}(W, C_2) \implies C_1 = C_2$$

Using the above dependencies, the following equalities are derived using the supplied instance of sum and count and the tuples in table 1.

1. $f(4, \{2\}, 6))$ = 2
2. $f(2, \{4\}, 6))$ = 4
3. $f(2, \{\}, 2))$ = $f(4, \{\}, 4))$ = 0 = $g(2, \{\}, 1))$ = $g(4, \{\}, 1))$
4. $g(4, \{2\}, 2))$ = 1 = $g(2, \{4\}, 2))$

Using the above equalities and the tuples of the predicate 'plus' in table 2, we leverage the inverse rules to expand average$(\{2, 4\}, S, C, A)$ in figure 6. In step 3 of the expansion, the functions $f(2, \{4\}, S)$ and $g(2, \{4\}, C)$ are substituted for the variables S_1 and C_1 respectively. Subsequently in step 6 of the expansion, the functions $f(4, \{\}, S_1)$ and $g(4, \{\}, C_1)$ are substituted for the variables S_2 and C_2 respectively. In addition the tuples of the 'plus' predicate are

1. average($\{2,4\}$, S, C, A)
2. average($\{4\}$, S_1, C_1, A_1), plus($S_1, 2, S$), plus($C_1, 1, C$), $A = \frac{S}{C}$
3. average($\{4\}$, S_1, C_1, A_1), plus($S_1, 2, S$), plus($C_1, 1, C$), $A = \frac{S}{C}$
4. average($\{4\}$, $f(2,\{4\},S)$, $g(2,\{4\},C)$, A_1), sum($\{2,4\}, S$), count($\{2,4\}C$), $A = \frac{S}{C}$
5. average($\{4\}$, $f(2,\{4\},S)$, $g(2,\{4\},C)$, A_1) ...
6. average($\{\}, S_2$, C_2, A_2), plus($S_2, 4, f(2,\{4\},S)$), plus($C_2, 1, g(2,\{4\},C)$), $A_1 = \frac{f(2,\{4\},S)}{g(2,\{4\},C)}$ ...
7. average($\{\}, f(4,\{\},4))$, $g(4,\{\},4))$, A_2), sum($\{4\}, f(2,\{4\},S)$), count($\{4\}, g(2,\{4\},C)$), $A_1 = \frac{f(2,\{4\},S)}{g(2,\{4\},C)}$ ...
8. average($\{\}, f(4,\{\},4))$, $g(4,\{\},4))$, A_2) .....

Figure 6: Reformulating average(W, A, S, C) using sum(W, S) and count(W, C)

reformulated in steps 3 and 6 using the inverse rules and the substitutions of S_1, S_2, C_1 and C_2.

The expansion of average$(\{2,4\}, S, C, A)$ which is shown in figure 6 leads to the derivation of $A = \frac{6}{2} = 3$ using the instances of the predicates sum and count. In the above expansion we compute the average of the input set by reformulating the plus operator using sum and count predicates. We note that the problem of deciding whether two recursive aggregate queries are equivalent is undecidable in general. However for a subset of recursive aggregate definitions the Invert$_{Agg}$ algorithm could potentially be leveraged to reformulate aggregate queries using views even when the relationship between aggregates in the query and the view (as in example 3) are not explicitly specified.

5 Related Work

The problem of reformulating database queries using views has been well-studied in the database community. A comprehensive survey of the techniques that are used to rewrite conjunctive queries using views is presented in (Halevy 2001). While the problem of rewriting queries using conjunctive views has received significant attention (Halevy 2001; Afrati and Chirkova 2011; Calvanese et al. 2003; Chirkova, Halevy, and Suciu 2002; Agrawal, Chaudhuri, and Narasayya 2000; Duschka and Genesereth 1997), only a small fraction of the work addresses the case where the query language contains aggregates (Cohen, Nutt, and Serebrenik 1999; Afrati, Li, and Ullman 2001; Agrawal, Chaudhuri, and Narasayya 2000; Gupta, Harinarayan, and Quass 1995; Srivastava et al. 1996). In addition, a common assumption among the techniques proposed in (Cohen, Nutt, and Serebrenik 1999; Afrati, Li, and Ullman 2001; Agrawal, Chaudhuri, and Narasayya 2000; Gupta, Harinarayan, and Quass 1995; Srivastava et al. 1996) is that the aggregates in the query and the view are the same. In contrast, we allow aggregates in our language to be expressed modularly in terms of other aggregates. We leverage this modularity (which is addressed in section 3.1) to reformulate aggregate queries using views where the aggregates in the query and the views are different.

The techniques proposed in (Afrati and Chirkova 2011; Afrati, Li, and Ullman 2001; Srivastava et al. 1996) reformulate aggregate queries using views in a restricted setting. For instance, the techniques proposed in (Cohen, Nutt, and Serebrenik 1999; Afrati and Chirkova 2011; Afrati, Li, and Ullman 2001) consider central rewritings only. In central rewritings only one view contributes to the aggregated value in the head. In contrast, our algorithm derives non-central reformulations of aggregate queries as well. Consider the reformulation which is presented in example 3. In example 3, we investigate whether the query q, which computes the average amount per cheque received by a female employee in the company XYZ, could be equivalently reformulated using the views v_1 and v_2, which record the annual salary of an employee and the number of cheques received by an employee respectively. Clearly such a reformulation exists. However, the reformulation is not a central rewriting since the aggregate in the head of the query is computed using aggregate values from two different views. In fact there are no central rewritings of the query q using the views v_1 and v_2. Previous work (Srivastava et al. 1996) has studied sufficient conditions for the usability of aggregate views to reformulate aggregate queries. However the conditions are specific to the aggregates min, max, sum and count only and in general do not address reformulations of user-defined aggregates in the query using other aggregates.

6 Conclusion

In this paper, we study the problem of reformulating aggregate queries using views in a LAV setting. Our proposal leverages the Inverse Method (Duschka and Genesereth 1997) to compute equivalent reformulations of an aggregate query using views where the aggregates in the query are expressed as views over the aggregates in the views. Although the problem of deciding equivalence of queries in general is undecidable, our algorithm computes equivalent reformulations when they exist. In addition, when the aggregates in the query and the views are different, our algorithm computes equivalent reformulations by leveraging modular aggregate definitions.

References

Afrati, F., and Chirkova, R. 2011. Selecting and using views to compute aggregate queries. *Journal Computer System Sciences*.

Afrati, F.; Li, C.; and Ullman, J. D. 2001. Generating efficient plans for queries using views. SIGMOD.

Agrawal, S.; Chaudhuri, S.; and Narasayya, V. R. 2000. Automated selection of materialized views and indexes in sql databases. VLDB.

Calvanese, D.; Giacomo, G. D.; Lenzerini, M.; and Vardi, M. Y. 2003. View-based query containment. PODS.

Chirkova, R.; Halevy, A. Y.; and Suciu, D. 2002. A formal perspective on the view selection problem. *The VLDB Journal*.

Cohen, S.; Nutt, W.; and Serebrenik, A. 1999. Rewriting aggregate queries using views. PODS.

Duschka, O. M., and Genesereth, M. R. 1997. Answering recursive queries using views. PODS.

Gupta, A.; Harinarayan, V.; and Quass, D. 1995. Aggregate-query processing in data warehousing environments. VLDB.

Halevy, A. Y. 2001. Answering queries using views: A survey. *The VLDB Journal*.

Srivastava, D.; Dar, S.; Jagadish, H. V.; and Levy, A. Y. 1996. Answering queries with aggregation using views. VLDB.

Ullman, J. D. 1989. *Principles of Database and Knowledge-Base Systems: Volume II*.