Algorithms for Rewriting Aggregate Queries Using Views

Sara Cohen[*] Werner Nutt[†] Alexander Serebrenik[*]

Abstract

Queries involving aggregation are typical in a number of database research areas, such as data warehousing, global information systems and mobile computing. One of the main ideas to optimize the execution of an aggregate query is to reuse results of previously answered queries. This leads to the problem of rewriting aggregate queries using views. More precisely, given a set of queries, called "views," and a new query, the task is to reformulate the new query with the help of the views in such a way that executing the reformulated query over the views yields the same result as executing the original query over the base relations. Due to a lack of theory, so far algorithms for this problem were rather ad-hoc. They were sound, but were not proven to be complete.

Previously we have given syntactic characterizations for the equivalence of aggregate queries, and applied them to decide when there exist rewritings. However, these decision procedures are nondeterministic and do not lend themselves immediately to an implementation. In the current paper, we refine those procedures by eliminating the nondeterminism as much as possible, thus obtaining practical algorithms for rewriting queries with the operators count and sum. It can be proved that our algorithms are complete for queries where each relation occurs only once and for queries without comparisons. We present an algorithm for rewriting nonaggregate queries that is more efficient than the known algorithms for this problem. We show how to modify this algorithm to obtain rewriting algorithms for queries with min and max. These algorithms are a basis for realizing optimizers that rewrite queries using views.

[*]Computer Science Department, The Hebrew University, Jerusalem 91904, Israel

[†]German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany

1 Introduction

Typically, queries over data warehouses are aggregate queries. Aggregate queries occur in different places in a data warehouse (DW). The ultimate purpose of a DW is to support queries by end users that want to analyze a business. They want to have a comprehensive picture of their company and ask for the number of customers, maximum sales, total profits, etc. However, aggregate queries are not only processed at the data warehouse back end. But also loading a data warehouse often requires computation of aggregates. Aggregation is a means of reducing the granularity of data. Data in a data warehouse is often coarser than the data in the operational systems that it is based on. For instance, while the operational data of a supermarket may contain each single item of a customers purchase, the DW may contain only the total sales of each item by day and store. In such a case, loading can be viewed as executing an aggregate query.

The execution of aggregate queries tends to be time consuming and costly. Computing one aggregate value often requires to scan many data items. This makes query optimization a necessity. A promising technique to speed up the execution of aggregate queries is to reuse the answers to previous queries to answer new queries. We call a reformulation of a query that uses other queries a *rewriting.* Finding such rewritings is known in the literature as the problem of *rewriting queries using views.* In this phrasing of the problem, it is assumed that there is a set of *views,* whose answers have been stored, or *materialized.* Then, given a query, the problem is to find a rewriting, which is formulated in terms of the views and some database relations, such that evaluating the original query yields the same answers as evaluating first the views and then, based on that result, evaluating the rewriting.

Rewriting techniques are not only applicable to query optimization, but also to data warehouse design [TS97], which in a sense is the inverse of the optimization problem. Here, one assumes that a set of queries over base relations is given, together with some additional information about the frequency with which the queries are posed and the frequency with which changes to the base relations occur. One is then interested in a set of intermediate views from which the queries can be computed such that the total cost of query execution and view maintenance is minimal. In order to find good views, one has to understand how queries can be rewritten using views.

Rewriting queries using views has been studied extensively for non-aggregate queries [LMSS95], and algorithms have been devised and implemented [LSK95, Qia96]. For aggregate queries, the problem has been investigated mainly in the special case of datacubes (see e.g., [HRU96, Dyr96]. However, there is little theory for general aggregate queries, and the rewriting algorithms that appear in the literature are by and large ad hoc. These algorithms are sound, that is, the reformulated queries they produce are in fact rewritings, but there is neither a guarantee that they output a rewriting whenever one exists, nor that they generate all existing rewritings [SDJL96, GHQ95].

Recently, we have given syntactic characterizations for the equivalence of SQL queries with the aggregate operators *min*, *max*, *count*, and *sum* [NSS98]. We have applied them to decide, given an aggregate query and a set of views, whether there exists a rewriting, and whether a new query over views and base relations is a rewriting [CNS99]. These characterizations, however, do not immediately yield practical algorithms.

In this paper, we show how to derive such algorithms. The algorithms are sound, i.e., they output rewritings. We can also show that they are complete for important cases, which are relevant in practice. In Section 2 we present a motivating example. A formal framework for rewritings of aggregate queries is presented in Section 3. In Section 4 we give algorithms for rewriting aggregate queries and in Section 5 we conclude.

2 Motivation

We discuss an example that illustrates the rewriting problem for aggregate queries. This example models exactly the payment policy for teaching assistants at the Hebrew University in Jerusalem. There are two tables with relations pertaining to salaries of teaching assistants:

```
ta(name, course_name, job_type)
salaries(job_type, sponsorship, amount).
```

At the Hebrew University, there may be many teaching assistants in a course. Each TA has a `job_type` in the course he assists. For example, he may give lectures or grade exercises. Teaching assistants are financed by different sources, like science foundations and the university itself. For each job type, each sponsor gives a fixed amount. Thus, a lab instructor may receive $600 per month from the university and $400 from a government science foundation.

We suppose that there are two materialized views. In the first view, `positions_per_type`, we compute the number of positions of each type held in the university:

```
CREATE VIEW positions_per_type(job_type, positions)
AS
SELECT    job_type, COUNT(*)
FROM      ta
GROUP BY  job_type
```

In the second view, `ta_salary` we compute the total salary for each type of position:

```
CREATE VIEW ta_salary(job_type, total_salary)
AS
SELECT    job_type, SUM(amount)
FROM      salaries
GROUP BY  job_type.
```

In the following query we calculate the total amount of money spent on each job position:

```
SELECT    ta.job_type, SUM(amount)
FROM      ta, salaries
WHERE     ta.job_type = salaries.job_type
GROUP BY  ta.job_type.
```

An intelligent query optimizer could now reason that for each type of job we can calculate the total amount of money spent on it if we multiply the salary that one TA receives for such a job by the number of positions of that type. The two materialized views contain information that can be combined to yield an answer to our query. The optimizer can formulate a new query that only accesses the views and does not touch the tables in the database:

```
SELECT    ta_salary.job_type, total_salary*positions
FROM      ta_salary, positions_per_type
WHERE     ta_salary.job_type =
                  positions_per_type.job_type
GROUP BY  ta_salary.job_type.
```

In order to evaluate the new query, we no more need to look up all the teaching assistants nor all the financing sources. Thus, probably, the new query can be executed more efficiently.

In this example, we used our common sense in two occasions. First, we gave an argument why evaluating the original query yields the same result as evaluating the new query that uses the views. Second, because we understood the semantics of the original query and the views, we were able to come up with a reformulation of the query over the views. Since we cannot expect a query optimizer to have common sense, we will only be able to build an optimizer that can rewrite aggregate queries, if we can provide answers to the following two questions.

- How can we prove that a new query, which uses views, produces the same results as the original query?

- How can we devise an algorithm that systematically and efficiently finds all rewritings?

If efficiency and completeness cannot be achieved at the same time, we may have to find a good trade-off between the two requirements.

3 A Formal Framework

In this section we define the formal framework in which we study rewritings of aggregate queries. First, we extend the well-known Datalog syntax for non-aggregate queries [Ull89] so that it covers also aggregates. This syntax is more abstract and concise than SQL. It is not only better suited for a theoretical investigation, but it is also a better basis for implementing algorithms that reason about queries, in particular for implementations in a logic programming language. Based on this syntax, we define the semantics of queries and give a precise formulation of the rewriting problem.

Through the syntax we implicitly define the set of SQL-queries to which our techniques apply. They are essentially nonnested queries without a `HAVING` clause and with the aggregate operators *min*, *max*, *count*, and *sum*. A generalization to queries with the constructor `UNION` is possible, but beyond the scope of this paper. For queries with arbitrary nesting and negation no rewriting algorithms are feasible, since equivalence of such queries is undecidable.

3.1 Non-Aggregate Queries

We recall the Datalog notation for conjunctive queries (which are also called select-project-join queries or SPJ queries) and fix our terminology. Later on, we will extend it to aggregate queries.

A *term* (denoted as s, t) is either a variable (denoted as x, y, z) or a constant. A *comparison* has the form $s_1 \ \rho \ s_2$, where ρ is one of $<$, $\leq$, $>$, and $\geq$.[1] If C and C' are conjunctions of comparisons, we write $C \models C'$ if C' is a *consequence* of C. In this paper we assume all comparisons range over the rationals.

We denote predicates as p, q, r. A *relational atom* has the form $p(s_1, \ldots, s_k)$. Sometimes we write $p(\bar{s})$, where $\bar{s}$ denotes the tuple of terms $s_1, \ldots, s_k$. An *atom* (denoted as a, b) is either a relational atom or a comparison.

A *conjunctive query* is an expression of the form

$$q(x_1, \ldots, x_k) \leftarrow a_1 \ \& \ \cdots \ \& \ a_n.$$

The atom $q(x_1, \ldots, x_k)$ is called the *head* of the query. The atoms $a_1, \ldots, a_n$ form the query *body*. They can be relational or comparisons. If the body contains no comparisons, then the query is *relational*. A query is *linear* if it does not contain two relational atoms with the same predicate symbol. We abbreviate a query as $q(\bar{x}) \leftarrow B(\bar{s})$, where $B(\bar{s})$ stands for the body and $\bar{s}$ for the terms occurring in

[1] We use the notation $s = t$ as abbreviation for the conjunction $s \leq t \ \& \ t \leq s$.

the body. Similarly, we may write a conjunctive query as $q(\bar{x}) \leftarrow R(\bar{s})\ \&\ C(\bar{t})$, in case we want to distinguish between the relational atoms and the comparisons in the body, or, shortly, as $q(\bar{x}) \leftarrow R\ \&\ C$. The variables appearing in the head are called *distinguished variables*, while those appearing only in the body are called *nondistinguished variables.* Atoms containing at least one nondistinguished variable are called *nondistinguished atoms.* By abuse of notation, we will often refer to a query by its head $q(\bar{x})$ or simply by the predicate of its head q.

A database $\mathcal{D}$ contains for each predicate a finite relation. Under *set semantics,* a conjunctive query q defines a new relation $q^{\mathcal{D}}$ which consists of all the answers that q produces over $\mathcal{D}$ (for a precise definition consult a standard textbook like [Ull89]). Under *bag-set semantics,* q defines a multiset or *bag* $\{\!\{q\}\!\}^{\mathcal{D}}$ of tuples for each database $\mathcal{D}$. The bag $\{\!\{q\}\!\}^{\mathcal{D}}$ contains the same tuples as the relation $q^{\mathcal{D}}$, but each tuple occurs as many times as it can be derived over $\mathcal{D}$ with q (for a precise definition see [CV93]).

Under set-semantics, two queries q and q' are *equivalent* if for every database, they return the same set as a result. Analogously, we define equivalence under bag-set-semantics.

In the example below, we show how to transform a query in SQL notation into one in Datalog notation.

Example 3.1 Consider a query that finds the teaching assistants who have a job for which they receive more then \$500 from the government:

```
SELECT  name
FROM    ta, salaries
WHERE   sponsorship = 'Govt.'  AND amount > 500
        AND ta.job_type = salaries.job_type.
```

We translate this query into an equivalent Datalog query with the head predicate q. For the relation names `ta` and `salaries` we introduce the predicate names ta and salaries, and for each attribute of a relation, we fix an argument position of the corresponding predicate. For each occurrence of a relation name in the `FROM` clause we create a relational atom. The selection constraints in the `WHERE` clause are taken into account by placing constants or identical variables into appropriate argument positions of the atoms corresponding to a relation, or by imposing comparisons on variables. Finally, the output arguments in the `SELECT` clause appear as the distinguished variables in the head.

$$\begin{aligned} \mathsf{q}(Name) \ \leftarrow\ & \mathsf{salaries}(Job\_Type, \texttt{'Govt.'}, Amount)\ \& \\ & \mathsf{ta}(Name, Course\_Name, Job\_Type)\ \& \\ & Amount > 500. \end{aligned}$$

It can be easily checked that the Datalog query q above is equivalent to our SQL query. Obviously, one notation can be transformed into the other, back and forth, completely automatically.

3.2 Aggregate Queries

We now extend the Datalog syntax so as to capture also queries with GROUP BY and aggregation. We restrict ourselves to SQL queries where the group by attributes are *identical* to those in the SELECT statement, although SQL only requires that the latter be a *subset* of those appearing in the GROUP BY clause. Also, we assume that queries have only one aggregate term. The general case can easily be reduced to this one.

Example 3.2 Recall the query in Section 2 that calculates the total amount of money spent on each job type. The extension of the Datalog syntax is straightforward. Since the SELECT attributes are identical to the grouping attributes, there is no need to single them out by a special notation. Hence, the only new feature is the aggregate term in the SELECT clause. We simply add it to the terms in the head of the query, after replacing the attributes with corresponding variables. Thus, the above SQL query is transformed into the following Datalog query:

$$\begin{aligned} &\mathsf{q}(\mathit{Job\_Type}, \mathit{sum}(\mathit{Amount})) \leftarrow \\ &\qquad \mathsf{ta}(\mathit{Name}, \mathit{Course\_Name}, \mathit{Job\_Type})\ \& \\ &\qquad \mathsf{salaries}(\mathit{Job\_Type}, \mathit{Sponsorship}, \mathit{Amount}). \end{aligned}$$

We now give a formal definition of the syntax and semantics of such aggregate queries. We are interested in queries with the aggregation functions *count*, *sum*, *min* and *max*. Since results for *min* are analogous to those for *max*, we do not consider *min*. Our function *count* is analogous to the function COUNT(*) of SQL.

An *aggregate term* is an expression built up using variables, the operations addition and multiplication, and aggregation functions.[2] For example, *count* and $\mathit{sum}(z_1 * z_2)$, are aggregate terms. We use κ as abstract notations for aggregate terms.

If we want to refer to the variables occurring in an aggregate term, we write $\kappa(\bar{y})$, where $\bar{y}$ is a tuple of distinct variables. Terms of the form *count*, $\mathit{sum}(y)$ and $\mathit{max}(y)$ are *elementary aggregate terms*. Abstractly, elementary aggregate terms are denoted as $\alpha(y)$, where α is an aggregate function.

An aggregate term $\kappa(\bar{y})$ naturally gives rise to a function $f_{\kappa(\bar{y})}$ that maps multisets of tuples of numbers to numbers. For instance, $\mathit{sum}(z_1 * z_2)$ describes the function $f_{\mathit{sum}(z_1 * z_2)}$ that maps any multiset M of pairs of numbers (m_1, m_2) to $\sum(m_1, m_2)$. We call such a function an *aggregation function*.

An *aggregate query* is a conjunctive query augmented by an aggregate term in its head. Thus, it has the form

$$q(\bar{x}, \kappa(\bar{y})) \leftarrow B(\bar{s}).$$

[2]This definition blurs the distinction between the function as a mathematical object and the symbol denoting the function. However, a notation that takes this difference into account would be cumbersome.

We call $\bar{x}$ the *grouping variables* of the query. Queries with elementary aggregate terms are *elementary queries*. If the aggregation term in the head of a query has the form $\alpha(y)$, we call the query an α-*query* (e.g., a max-query).

In this paper we are interested in rewriting elementary queries using elementary views. However, as the example in Section 2 shows, even under this restriction the rewritings may not be elementary.

We now give a formal definition of the semantics of aggregate queries. Consider the query $q(\bar{x}, \kappa(\bar{y})) \leftarrow B(\bar{s})$. For a database $\mathcal{D}$, the query yields a new relation $q^{\mathcal{D}}$. To define the relation $q^{\mathcal{D}}$, we proceed in two steps.

We associate to q a non-aggregate query, $\breve{q}$, called the *core* of q, which is defined as $\breve{q}(\bar{x}, \bar{y}) \leftarrow B(\bar{s})$. The core is the query that returns all the values that are amalgamated in the aggregate. Recall that under bag-set-semantics, the core returns over $\mathcal{D}$ a bag $\{\!\{\breve{q}\}\!\}^{\mathcal{D}}$ of tuples $(\bar{d}, \bar{e})$. For a tuple of constants $\bar{d}$ of the same length as $\bar{x}$ let

$$,_{\bar{d}} \;:=\; \left\{\!\!\left\{ \bar{e} \;\middle|\; (\bar{d}, \bar{e}) \in \{\!\{\breve{q}\}\!\}^{\mathcal{D}} \right\}\!\!\right\}.$$

That is, the bag $,_{\bar{d}}$ is obtained by first grouping together those answers to $\breve{q}$ that return $\bar{d}$ for the grouping terms, and then stripping off from those answers the prefix $\bar{d}$. In other words, $,_{\bar{d}}$ is the multiset of $\bar{y}$-values that $\breve{q}$ returns for $\bar{d}$.

Now we define the result of evaluating q over $\mathcal{D}$ as

$$q^{\mathcal{D}} \;:=\; \{(\bar{d}, e) \mid ,_{\bar{d}} \neq \emptyset \text{ and } e = f_{\kappa(\bar{y})}(,_{\bar{d}})\},$$

that is, intuitively, whenever there is a nonempty group of answers with index $\bar{d}$, then we apply the aggregation function $f_{\kappa(\bar{y})}$ to the multiset of $\bar{y}$-values of that group.

Again, two aggregate queries q and q' are *equivalent* if $q^{\mathcal{D}} = q'^{\mathcal{D}}$ for all databases $\mathcal{D}$.

3.3 Equivalence Modulo a Set of Views

Up to now, we have defined equivalence of aggregate queries and equivalence of non-aggregate queries under both, set and bag-set-semantics. Under set semantics, equivalence of conjunctive queries can be decided by checking whether there exist *containment mappings* or *homomorphisms* between the queries [CM77, JK83].

However, the relationship between a query q and a rewriting r of q is not equivalence of queries, because the view predicates occurring in r are not regular data base relations, but are determined by the base relations indirectly. In order to take this relationship into account, we give a definition of equivalence of queries modulo a set of views.

We consider aggregate queries that use predicates both from $\mathcal{R}$, a set of base relations, and $\mathcal{V}$, a set of view definitions. We want to define the result of evaluating such a query over a database $\mathcal{D}$. We assume that a database contains only facts about the base relations.

For a database $\mathcal{D}$, let $\mathcal{D}_{\mathcal{V}}$ be the database that extends $\mathcal{D}$ by interpreting every view predicate $v \in \mathcal{V}$ as the relation $v^{\mathcal{D}}$. If q is a query that contains also predicates from $\mathcal{V}$, then $q^{\mathcal{D}_{\mathcal{V}}}$ is the relation that results from evaluating q over the extended database $\mathcal{D}_{\mathcal{V}}$. If q, q' are two aggregate queries using predicates from $\mathcal{R} \cup \mathcal{V}$, we define that q and q' are *equivalent modulo* $\mathcal{V}$, written $q \equiv_{\mathcal{V}} q'$, if $q^{\mathcal{D}_{\mathcal{V}}} = q'^{\mathcal{D}_{\mathcal{V}}}$ for all databases $\mathcal{D}$.

3.4 General Definition of Rewriting

Our goal is to rewrite an aggregate query using a set of views. We first give a general definition of rewritings. Later on, we will concentrate on rewritings that have a special form. Let q be a query, $\mathcal{V}$ be a set of views over the set of relations $\mathcal{R}$, and r be a query over $\mathcal{V} \cup \mathcal{R}$. All of q, r, and the views in $\mathcal{V}$ may be aggregate queries or not. Then we say that r is a *rewriting* of q using $\mathcal{V}$ if $q \equiv_{\mathcal{V}} r$ and r contains only atoms with predicates from $\mathcal{V}$. If $q \equiv_{\mathcal{V}} r$ and r contains at least one atom with a predicate from $\mathcal{V}$ we say that r is a *partial rewriting of q using* $\mathcal{V}$.

Now we can reformulate the intuitive questions we asked in the end of the Section 2. Given queries q and r, and a set of views $\mathcal{V}$, check whether $q \equiv_{\mathcal{V}} r$. Given a query q and a set of views $\mathcal{V}$, find all (some) rewritings or partial rewritings of q.

4 Rewritings of Aggregate Queries

We now present techniques for rewriting aggregate queries. Our approach will be a to generalize the known techniques for conjunctive queries. Therefore, we first give a short review of the conjunctive case and then discuss in how far aggregates give rise to more complications.

4.1 Rewritings of Relational Conjunctive Queries

We review the questions related to rewriting relational conjunctive queries. Suppose, we are given a set of conjunctive queries $\mathcal{V}$, the views, and another conjunctive query q. We want to know whether there is a rewriting of q using the views in $\mathcal{V}$.

The *first question* that arises is, what is the *language* for expressing rewritings? Do we consider arbitrary first order formulas over the view predicates as candidates, or even recursive queries, or do we restrict ourselves to conjunctive queries over the views? Since reasoning about queries in the first two languages is undecidable, researchers have only considered conjunctive rewritings.[3] Thus, a candidate for a rewriting of $q(\bar{x})$ has the form

$$r(\bar{x}) \ \leftarrow \ v_1(\theta_1\bar{x}_1) \ \& \ \ldots \ \& \ v_n(\theta_n\bar{x}_n),$$

where the θ_i's are substitutions that instantiate the view predicates $v_i(\bar{x}_i)$.

The *second question* is whether we can *reduce reasoning* about the query r, which contains view predicates, to reasoning about a query that has only base predicates. To this end, we *unfold* r. That is, we replace each view atom $v_i(\theta_i\bar{x}_i)$, with the instantiation $\theta_i B_i$ of the body of v_i, where v_i is defined as $v_i(\bar{x}_i) \leftarrow B_i$. We assume that the nondistinguished variables in different bodies are distinct. We thus obtain the unfolding r^u of r, for which the Unfolding Theorem holds:

$$r^u(\bar{x}) \leftarrow \theta_1 B_1 \ \& \ \ldots \ \& \ \theta_n B_n.$$

Theorem 4.1 (Unfolding Theorem) *Let $\mathcal{V}$ be a set of views, r a query over $\mathcal{V}$, and r^u be the unfolding of r. Then r and r^u are equivalent modulo $\mathcal{V}$, that is,*

$$r \equiv_{\mathcal{V}} r^u. \tag{1}$$

The *third question* is how to check whether r is a rewriting of q, that is, whether r and q are *equivalent modulo $\mathcal{V}$*. Because of the Unfolding Theorem, this can be achieved by checking whether r^u and q are set-equivalent: if $r^u \equiv q$, then (1) implies $r \equiv_{\mathcal{V}} q$. Set-equivalence of conjunctive queries can be decided syntactically by checking whether there are homomorphisms in both directions.

4.2 Rewritings of Count-queries

We consider the problem of rewriting *count*-queries. As a first step, we consider rewriting relational *count*-queries. We then extend our technique in order to rewrite *count*-queries with comparisons.

4.2.1 Rewritings of Relational Count-queries

When rewriting *count*-queries, we must deal with the same questions that arose when rewriting conjunctive queries. Thus, we first define the language for expressing rewritings. Even if we restrict the language to conjunctive aggregate queries over the views, we still must decide on two additional issues. First, which types of aggregate views are useful for a rewriting? Second, what will be the

[3]It is an interesting theoretical question, which as yet has not been resolved, whether more expressive languages give more possibilities for rewritings. It is easy to show, at least, that in the case at hand allowing also disjunctions of conjunctive queries as candidates does not give more possibilities than allowing only conjunctive queries.

aggregation term in the head of the rewriting? A *count*-query is sensitive to multiplicities, and *count*-views are the only type of aggregate views that do not lose multiplicities[4]. Thus, the natural answer to the first question is to use only *count*-views when rewriting *count*-queries. We show in the following example that there are an infinite number of aggregate terms that can be usable in rewriting a *count*-query.

Example 4.2 Consider aggregate queries q, v, and rewriting candidates, r_n, for $n \geq 1$:

$$\begin{array}{rcl} q(x, \mathit{count}) & \leftarrow & p(x,y) \\ v(x, \mathit{count}) & \leftarrow & p(x,y) \\ r_n(x, (\prod_{j=1}^{n} z_j)^{\frac{1}{n}}) & \leftarrow & v(x,z_1) \ \& \ \ldots \ \& \ v(x,z_n) \end{array}$$

Then it is easy to see that r_n is a rewriting of q modulo $\{v\}$ for all n. It is natural to create only r_1 as a rewriting of q. In fact, only for r_1 will the Unfolding Theorem hold. However, when creating rewritings automatically we must restrict the aggregate term in the head of the rewriting in order to prevent deriving infinitely many rewritings.

We define a candidate for a rewriting of $q(\bar{x}, \mathit{count})$ as a query having the form

$$r(\bar{x}, \mathit{sum}(\prod_{i=1}^{n} z_i)) \leftarrow v_1^c(\theta_1 \bar{x}_1, z_1) \ \& \ \ldots \ \& \ v_n^c(\theta_n \bar{x}_n, z_n)$$

where v_i^c are *count*-views defined as $v_i^c(\bar{x}_i, \mathit{count}) \leftarrow B_i$ and z_i are variables not appearing elsewhere in the body of r. We call r a *count*-rewriting candidate. Note that in some cases, it is possible to omit the summation. This is true if the values of z_i are functionally dependent on the value of $\bar{x}$. In such a case, the summation is always over a singleton.

After presenting our rewriting candidates we now show how we can reduce reasoning about rewriting candidates, to reasoning about conjunctive aggregate queries. We use a similar technique to that shown in Subsection 4.1. We replace view atoms with the appropriate instantiations of their bodies. As in the case for relational queries, unfolding should preserve equivalence. Otherwise reduction about the reasoning is not possible. We choose to replace the aggregate term in the rewriting with the *count* term. This is a natural choice and is also necessary since *count* is the only aggregate term which will preserve this characteristic. Thus, we obtain the unfolding r^u of r defined as

$$r^u(\bar{x}, \mathit{count}) \leftarrow \theta_1 B_1 \ \& \ \ldots \ \& \ \theta_n B_n.$$

[4]Although *sum*-views are sensitive to multiplicities (i.e., are calculated under bag-set-semantics), they lose these values. For example, *sum*-views ignore occurrences of zero values.

We have proven in [CNS99] that the unfolding theorem holds, i.e., $r \equiv_{\mathcal{V}} r^u$. Moreover, we have shown that when unfoldings of this type are used, the candidate we defined is the only candidate preserving this characteristic. Now, instead of checking whether r is a rewriting of q, we can verify if r^u is equivalent to r. It has been shown [CV93, NSS98] that two relational *count*-queries are equivalent if and only if they are isomorphic.

We present an algorithm that finds a rewriting for a query using views. Our approach can be thought of as reverse engineering. We have characterized the "product" that we must create, i.e., a rewriting, and we now present an automatic technique for producing it.

We start by discussing when a view, $v(\bar{u}, count) \leftarrow R_v$, instantiated by θ, is usable in order to rewrite a query, $q(\bar{x}, count) \leftarrow R$. Recall that a rewriting of q is a query r that when unfolded yields a query isomorphic to q. Thus, in order for θv, to be usable, θR_v must "cover" some part of R. Therefore, θv is usable for rewriting q only if there exists an isomorphism, φ, from R_v to $R' \subseteq R$. Note that we can assume, w.l.o.g. that φ is the identity mapping on the distinguished variables of v. We would like to replace R' with θv in the body of q in order to derive a partial rewriting of q. This cannot always be done. Observe that after replacing R' with θv, variables that appeared in R' and do not appear in $\theta \bar{u}$ are not accessible anymore. Thus, we can only perform the replacement if these variables do not appear anywhere else in q, in q's head or body. If R' can be replaced by θv, using φ, we say that *v is R-usable under θ w.r.t. φ* to rewrite q. We denote this fact as R-$\mathsf{usable}(v,\ \theta,\ \varphi)$.

Example 4.3 Consider the following

$$
\begin{aligned}
q(x, count) &\leftarrow p_1(x, x, y) \,\&\, p_2(y, z) \\
v(x', u', count) &\leftarrow p_1(x', u', y')
\end{aligned}
$$

In order to use v in rewriting q we must find an instantiation θ such that $\theta p_1(x', u', y')$ covers some part of the body of q. Clearly, $\theta p_1(x', u', y')$ can cover only $p_1(x, x, y)$. Thus, θ should equate x' and u'. We take, $\theta = \{x'/x, u'/x\}$ and thus, $\varphi = \{x/x, y'/y\}$. However, y appears in $p_1(x, x, y)$ and not in the head of θv and therefore, y is not accessible after replacement. Note that, y appears in p_2 and thus, v is not R-usable in rewriting q.

In Figure 4.2.1 we present an algorithm for finding rewritings. The algorithm nondeterministically chooses a view v and an instantiation θ, such that v is R-usable under θ. If the choice fails, backtracking is performed. When the while-loop is completed, the algorithm returns a rewriting. By backtracking we can find additional rewritings. Note, that the same algorithm may be used to produce partial rewritings. For this purpose it is sufficient to relax the termination condition of the while-loop. This will similarly hold for subsequent algorithms presented.

```
Algorithm   Relational_Count_Rewriting
Input       A query $q(\bar{x}, count) \leftarrow R$ and a set of views $\mathcal{V}$
Output      A rewriting $r$ of $q$.

(1)   $Not\_Covered := R$.
(2)   $Rewriting := \emptyset$.
(3)   $n := 0$.
(4)   While $Not\_Covered \neq \emptyset$ do:
(5)       Choose a view $v(\bar{x}', count) \leftarrow R'$ in $\mathcal{V}$.
(6)       Choose an instantiation, $\theta$, and an isomorphism $\varphi$,
              such that $R$-usable($v$, $\theta$, $\varphi$).
(7)       For each atom $a \in R'$ do:
(8)           If $a$ is a nondistinguished atom, then
(9)               Remove $\varphi(\theta a)$ from $R$.
(10)              If $\varphi(\theta a) \notin Not\_Covered$ then fail.
(11)          Remove $\varphi(\theta a)$ from $Not\_Covered$.
(12)      Increment $n$.
(13)      Add $v(\theta\bar{x}', z_n))$ to $Rewriting$, where $z_n$ is a fresh variable.
(14)  Return $r(\bar{x}, sum(\prod_{i=1}^{n} z_i)) \leftarrow Rewriting$.
```

Figure 1: Relational Count Query Rewriting Algorithm

We note the following. In line 9 R is changed and thus, q is also changed. Therefore, at the next iteration of the while-loop we check whether v is R-usable under θ to rewrite the updated version of q (line 6). Thus, in each iteration of the loop, additional atoms are covered. In line 10 the algorithm checks if a nondistinguished atom is already covered. If so, then the algorithm must fail, i.e., backtrack, as explained above. The algorithm is sound and complete as stated below.

Theorem 4.4 (Soundness and Completeness of Relational Count Rewriting) *Let q be a count-query and $\mathcal{V}$ be a set of views. Then r is a count-rewriting candidate and $r \equiv_{\mathcal{V}} q$ if and only if r can be returned by the call* Relational_Count_Rewriting$(q, \mathcal{V})$*, by making the appropriate choices in lines 5 and 6.*

Our algorithm runs in nondeterministic polynomial time. The algorithm guesses views and instantiations and then verifies that the obtained result is a rewriting in a polynomial time. This is an optimal algorithm, since the view usability problem for relational conjunctive *count*-queries is NP-complete [CNS99].

4.2.2 Rewritings of Count-Queries with Comparisons

We extend the technique presented in the previous section in order to rewrite queries with comparisons. Thus, we consider the problem of rewriting queries having comparisons with views having comparisons. We augment the rewriting candidate form with comparisons. Thus given a query $q(\bar{x}, count) \leftarrow R \ \& \ C$ and a set of views, $\mathcal{V}$, a rewriting candidate has the form:

$$\begin{aligned} r(\bar{x}, sum(\prod_{i=1}^{n} z_i)) \leftarrow & \\ & v_1^c(\theta_1\bar{x}_1, z_1) \ \& \ \ldots \ \& \ v_n^c(\theta_n\bar{x}_n, z_n) \ \& \ C' \end{aligned}$$

We can unfold r in the same fashion as unfolding a relational rewriting. As above, it holds that $r^u \equiv_{\mathcal{V}} r$. Thus, once again we can reduce reasoning about queries with views to reasoning about equivalent queries without views. In order to verify that r is a rewriting of q , we have to verify that $r^u \equiv q$.

In [NSS98], we gave a sound and complete characterization of equivalence of conjunctive *count*-queries. The only known algorithm that checks equivalence of conjunctive *count*-queries creates an exponential blowup of the queries. Thus, it is difficult to present a tractable algorithm for computing rewritings. However, equivalence of linear *count*-queries with comparisons is isomorphism of the queries [NSS98]. Thus, we will give a sound, complete, and tractable algorithm for computing rewritings of linear *count*-queries. This algorithm is also sound and tractable for the general case, but is not complete. In the sequel, in this section we assume that the comparisons in the queries we are rewriting are deductively closed. Computing a deductive closure is a well-known polynomial procedure [Klu88].

We discuss when a view, $v(\bar{u}, count) \leftarrow R_v \ \& \ C_v$, instantiated by θ, is usable in order to rewrite a query, $q(\bar{x}, count) \leftarrow R \ \& \ C$. Clearly, the conditions presented above for the relational case must hold in this case. Thus, in order for θv to be usable there must be an isomorphism φ from R_v to R', a subset of R. In addition we must require that $C \models \varphi(\theta C_v)$, thus, using v will preserve the comparisons implied by q. We have seen that when replacing R' with θv we lose access to the nondistinguished variables in v. Therefore, it is necessary for the comparisons in v to imply all the comparisons in q which contain an image of a nondistinguished variable in v. Formally, let $ndv(v)$ be the set of nondistinguished variables in v. Let $C^{\varphi(\theta ndv(v))}$ be the comparisons in C containing variables in $\varphi(\theta ndv(v))$. Then $C_v \models C^{\varphi(\theta ndv(v))}$ must hold. If these conditions hold, we say that v *is* C*-usable under* θ *w.r.t.* φ. We denote this fact as C-usable(v, θ, φ).

We present an algorithm for computing rewritings of conjunctive *count*-queries in Figure 4.2.2. Note that we modify C in line 12. We remove from C its comparisons containing a variable that is not accessible after replacing the appropriate subset of R by the appropriate instantiation of v. Thus, this step is necessary in order for the resulting query to be safe [Ull88].

```
Algorithm    Count_Rewriting
Input        A query $q(\bar{x}, count) \leftarrow R \,\&\, C$ and a set of views $\mathcal{V}$
Output       A rewriting $r$ of $q$.

 (1)   $Not\_Covered := R$.
 (2)   $Rewriting := \emptyset$.
 (3)   $n := 0$.
 (4)   While $Not\_Covered \neq \emptyset$ do:
 (5)       Choose a view $v(\bar{x}', count) \leftarrow R' \,\&\, C'$ in $\mathcal{V}$.
 (6)       Choose an instantiation, $\theta$, and an isomorphism $\varphi$,
               such that $R$-usable($v$, $\theta$, $\varphi$) and $C$-usable($v$, $\theta$, $\varphi$).
 (7)       For each atom $a \in R'$ do:
 (8)           If $a$ is a nondistinguished atom, then
 (9)                   Remove $\varphi(\theta a)$ from $R$.
(10)                   If $\varphi(\theta a) \notin Not\_Covered$ then fail.
(11)           Remove $\varphi(\theta a)$ from $Not\_Covered$.
(12)       Remove from $C$ comparisons containing a variable in $\varphi(\theta R')$,
               but not in $\theta\bar{x}'$
(13)       Increment $n$.
(14)       Add $v(\theta\bar{x}', z_n))$ to $Rewriting$, where $z_n$ is a fresh variable.
(15)   Return $r(\bar{x}, sum(\prod_{i=1}^{n} z_i)) \leftarrow Rewriting \,\&\, C$.
```

Figure 2: Count Query Rewriting Algorithm

The given algorithm runs in nondeterministic polynomial time. The following theorem states that it is both sound and complete for linear queries and is sound, but not complete, for arbitrary queries.

Theorem 4.5 (Soundness and Completeness of Count Rewriting) *Let q be a count-query and $\mathcal{V}$ be a set of views. If r is returned by* Count_Rewriting$(q, \mathcal{V})$*,* *then r is a count-rewriting candidate of q and $r \equiv_{\mathcal{V}} q$. In addition, if q is either linear or relational, then the opposite holds by making the appropriate choices in lines 5 and 6.*

Example 4.6 This example shows the incompleteness of the algorithm for the general case. Consider the query q, view v, rewriting r, and unfolding r^u

$$\begin{aligned} q(count) &\leftarrow p(x) \mathbin{\&} p(y) \mathbin{\&} p(u) \mathbin{\&} \\ &\quad\; x < y \mathbin{\&} x < u \\ v(y, count) &\leftarrow p(x) \mathbin{\&} p(y) \mathbin{\&} x < y \\ r(sum(z_1 * z_2)) &\leftarrow v(y, z_1) \mathbin{\&} v(y, z_2) \\ r^u(count) &\leftarrow p(x) \mathbin{\&} p(y) \mathbin{\&} p(u) \mathbin{\&} \\ &\quad\; x < y \mathbin{\&} u < y. \end{aligned}$$

Although $r^u \equiv q$ [NSS98], the algorithm does not find any rewritings.

4.3 Rewritings of Sum-Queries

Rewriting *sum*-queries is similar to rewriting *count*-queries. When rewriting *sum*-queries we must also take the aggregation variable into consideration. We present an algorithm for rewriting *sum*-queries that is similar to the algorithm for *count*-queries.

We define the form of rewriting candidates for *sum*-queries. Since *sum* and *count*-views are the only views that are sensitive to multiplicities, they are useful for rewritings. However, *sum*-views may lose multiplicities and make the aggregation variable inaccessible. Thus, at most one *sum*-view should be used in the rewriting of a query. The following are rewriting candidates of the query $q(\bar{x}, sum(y)) \leftarrow R \mathbin{\&} C$:

$$r_1(\bar{x}, sum(y * \prod_{i=1}^{n} z_i)) \leftarrow \tag{2}$$
$$v_1^c(\theta_1\bar{x}_1, z_1) \mathbin{\&} \ldots \mathbin{\&} v_n^c(\theta_n\bar{x}_n, z_n) \mathbin{\&} C'$$
$$r_2(\bar{x}, sum(y * \prod_{i=1}^{n} z_i)) \leftarrow v^s(\theta_s\bar{x}_s, y) \mathbin{\&} \tag{3}$$
$$v_1^c(\theta_1\bar{x}_1, z_1) \mathbin{\&} \ldots \mathbin{\&} v_n^c(\theta_n\bar{x}_n, z_n) \mathbin{\&} C'$$

where v_i^c is a *count*-view of the form $v_i^c(\bar{x}_i, count) \leftarrow B_i$ and v^s is a *sum*-view of the form $v^s(\bar{x}_s, sum(y)) \leftarrow B_s$. Note that the variable y in the head of the query in Equation 3 must appear among $\theta_i\bar{x}_i$ for some i. In [CNS99] we showed that if a rewriting candidate is equivalent to its unfolding then it must be one of the above forms. As in the case of *count*-query rewritings, in some cases the rewriting may be optimized by dropping the summation.

Once again, we reduce reasoning about rewriting candidates to reasoning about conjunctive aggregate queries. For this purpose we extend the unfolding technique introduced in Subsection 4.2. Thus, the unfoldings of the candidates presented are:

$$\begin{aligned} r_1^u(\bar{x}, sum(y)) &\leftarrow \theta_1 B_1 \;\&\; \ldots \theta_n B_n \;\&\; C'. \\ r_2^u(\bar{x}, sum(y)) &\leftarrow \theta_s B_s \;\&\; \theta_1 B_1 \;\&\; \ldots \theta_n B_n \;\&\; C'. \end{aligned}$$

Now, instead of checking whether r is a rewriting of q we can verify if r^u is equivalent to r. However, the only known algorithm for checking equivalence of *sum*-queries, presented in [NSS98], requires an exponential blowup of the queries. Thus, it might be very difficult to provide a tractable algorithm that is both sound and complete for arbitrary *sum*-queries. However, relational *sum*-queries and linear *sum*-queries are equivalent if and only if they are isomorphic. Thus, we can extend the algorithm presented in the Figure 4.2.2 for *sum*-queries.

As a preliminary step for our algorithm we extend the algorithm in Figure 4.2.2, such that in line 5 *sum*-views may be chosen as well. We call this algorithm Compute_Rewriting. We derive an algorithm for rewriting *sum*-queries, presented in Figure 4.3. The algorithm runs in nondeterministic polynomial time and the following holds:

```
Algorithm   Sum_Rewriting
Input       A query $q(\bar{x}, sum(y)) \leftarrow B$ and a set of views $\mathcal{V}$
Output      A rewriting $r$ of $q$.

(1)   Let $q'$ be the query $q'(\bar{x}, count) \leftarrow B$.
(2)   Let $r'$=Compute_Rewriting($q', \mathcal{V}$).
(3)   If $r'$ is of the form
          $r'(\bar{x}, sum(y * \prod_{i=1}^{n} z_i)) \leftarrow v^s(\theta_s \bar{x}_s, y) \;\&\; v_1^c(\theta_1 \bar{x}_1, z_1) \;\&\; \ldots \;\&$
                                            $v_n^c(\theta_n \bar{x}_n, z_n) \;\&\; C'$
(4)   Then return $r'$
(5)   If $r'$ is of the form
          $r'(\bar{x}, sum(\prod_{i=1}^{n} z_i)) \leftarrow v_1^c(\theta_1 \bar{x}_1, z_1) \;\&\; \ldots \;\&\; v_n^c(\theta_n \bar{x}_n, z_n) \;\&\; C'$
          and $y$ appears among $\theta_i \bar{x}_i$
(6)   Then return
          $r(\bar{x}, sum(y * \prod_{i=1}^{n} z_i)) \leftarrow v_1^c(\theta_1 \bar{x}_1, z_1) \;\&\; \ldots \;\&\; v_n^c(\theta_n \bar{x}_n, z_n) \;\&\; C'$.
```

Figure 3: Sum Query Rewriting Algorithm

Theorem 4.7 (Soundness and Completeness of Sum Rewriting) *Given a sum-query q and a set of views $\mathcal{V}$ the following holds. If r is returned by* $\mathsf{Sum\_Rewriting}(q, \mathcal{V})$*, then r is a sum-rewriting candidate of q and $r \equiv_{\mathcal{V}} q$. In addition, if q is either linear or relational, then the opposite holds by making the appropriate choices.*

4.4 Rewritings of Max-Queries

We consider the problem of rewriting max-queries. Note that max-queries are insensitive to multiplicities. Thus, it is natural to use nonaggregate views and max-views when rewriting a max-query. When using a max-view the aggregation variable becomes inaccessible. Thus, we use at most one max-view. The following are rewriting candidates of the query q:

$$\begin{aligned} r_1(\bar{x}, max(y)) \leftarrow& \qquad\qquad (4)\\ & v_1(\theta_1\bar{x}_1) \;\&\; \dots \;\&\; v_n(\theta_n\bar{x}_n) \;\&\; C' \\ r_2(\bar{x}, max(y)) \leftarrow& \; v^s(\theta_m\bar{x}_m, y) \;\& \qquad\qquad (5)\\ & v_1(\theta_1\bar{x}_1) \;\&\; \dots \;\&\; v_n(\theta_n\bar{x}_n) \;\&\; C' \end{aligned}$$

where v_i is a nonaggregate view and v^m is a max-view. Note that the variable y in the head of the query in Equation 5 must appear among $\theta_i\bar{x}_i$ for some i. In [CNS99] we showed that if a rewriting candidate is equivalent to its unfolding then it must have one of the above forms.

Once again, reasoning about rewriting candidates can be reduced to reasoning about max-queries, using an appropriate extension of the unfolding technique. We have shown [NSS98] that equivalence of relational max-queries is equivalence of their cores. There is a similar reduction for the general case. Thus, algorithms developed for checking set-equivalence of queries can easily be converted to algorithms for checking equivalence of max-queries. Similarly, algorithms that find rewritings of nonaggregate queries can be modified to find rewritings of max-queries. Rewriting nonaggregate queries is a well known problem [LMSS95]. Thus, we do not present algorithms for finding rewritings of max-queries in this paper.

5 Conclusion

Aggregate queries are increasingly prevalent due to the widespread use of data warehousing for decision support. They are generally computationally expensive since they scan many data items, while retrieving few. Thus, the computation time of aggregate queries is generally orders of magnitude larger than the result size of the query. This makes query optimization a necessity. Optimizing aggregate queries was studied in the context of datacubes [HRU96, Dyr96]. However, there was little theory for general aggregate queries, beyond this context. In this paper, based on previous results in [NSS98, CNS99], we presented algorithms that enable reuse of precomputed queries in answering new ones. The algorithms presented were implemented in SICStus Prolog.

Topics for future research include rewriting queries with `HAVING` clauses, negation and functional dependencies, and enriching the class of aggregate functions with statistical functions.

Acknowledgement

This research was supported by the Esprit Long Term Research Project 22469 "Foundations of Data Warehouse Quality" (DWQ).

References

[CM77] A.K. Chandra and P.M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proc. 9th Annual ACM Symposium on Theory of Computing*, 1977.

[CNS99] S. Cohen, W. Nutt, and A. Serebrenik. Rewriting aggregate queries using views. In Ch. Papadimitriou, editor, *Proc. 18th Symposium on Principles of Database Systems*, Philadelphia (Pennsylvania, USA), May 1999. ACM Press. To appear.

[CV93] S. Chaudhuri and M. Vardi. Optimization of real conjunctive queries. In *Proc. 12th Symposium on Principles of Database Systems*, Washington (D.C., USA), May 1993. ACM Press.

[Dyr96] C. Dyreson. Information retrieval from an incomplete datacube. In *Proc. 22nd International Conference on Very Large Data Bases*, Bombay (India), September 1996. Morgan Kaufmann Publishers.

[GHQ95] A. Gupta, V. Harinarayan, and D. Quass. Aggregate query processing in data warehouses. In *Proc. 21st International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers, August 1995.

[HRU96] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. In *Proc. 1996 ACM SIGMOD International Conference on Management of Data*, pages 205–227, Montreal (Canada), June 1996.

[JK83] D.S. Johnson and A. Klug. Optimizing conjunctive queries that contain untyped variables. *SIAM Journal on Computing*, 12(4):616–640, 1983.

[Klu88] A. Klug. On conjunctive queries containing inequalities. *J. ACM*, 35(1):146–160, 1988.

[LMSS95] A.Y. Levy, A.O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proc. 14th Symposium on Principles of Database Systems*, pages 95–104, San Jose (California, USA), May 1995. ACM Press.

[LSK95] A.Y. Levy, D. Srivastava, and T. Kirk. Data model and query evaluation in global information systems. *Journal of Intelligent Information Systems*, 5(2):121–143, 1995.

[NSS98] W. Nutt, Y. Sagiv, and S. Shurin. Deciding equivalences among aggregate queries. In J. Paredaens, editor, *Proc. 17th Symposium on*

Principles of Database Systems, pages 214–223, Seattle (Washington, USA), June 1998. ACM Press. Long version as Report of Esprit LTR DWQ.

[Qia96] X. Qian. Query folding. In Stanley Y. W. Su, editor, *Proc. 12th International Conference on Data Engineering*, pages 48–55, New Orleans (Louisiana, USA), March 1996. IEEE Computer Society.

[SDJL96] D. Srivastava, Sh. Dar, H.V. Jagadish, and A.Y. Levy. Answering queries with aggregation using views. In *Proc. 22nd International Conference on Very Large Data Bases*, Bombay (India), September 1996. Morgan Kaufmann Publishers.

[TS97] D. Theodoratos and T.K. Sellis. Data warehouse configuration. In *Proc. 23nd International Conference on Very Large Data Bases*, pages 126–135, Athens (Greece), August 1997. Morgan Kaufmann Publishers.

[Ull88] J. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. I: Classical Database Systems.* Computer Science Press, New York (New York, USA), 1988.

[Ull89] J. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. II: The New Technologies.* Computer Science Press, New York (New York, USA), 1989.