Rewriting Aggregate Queries Using Views

Sara Cohen
Institute for Computer Science
The Hebrew University
Jerusalem 91904, Israel
sarina@cs.huji.ac.il

Werner Nutt
German Research Center for
Artificial Intelligence GmbH
Stuhlsatzenhausweg 3
66123 Saarbrücken, Germany
Werner.Nutt@dfki.de

Alexander Serebrenik
Institute for Computer Science
The Hebrew University
Jerusalem 91904, Israel
alicser@cs.huji.ac.il

Abstract

We investigate the problem of rewriting queries with aggregate operators using views that may or may not contain aggregate operators. A rewriting of a query is a second query that uses view predicates such that evaluating first the views and then the rewriting yields the same result as evaluating the original query. In this sense, the original query and the rewriting are equivalent modulo the view definitions. The queries and views we consider correspond to unnested SQL queries, possibly with union, that employ the operators *min*, *max*, *count*, and *sum*.

Our approach is based on syntactic characterizations of the equivalence of aggregate queries. One contribution of this paper are characterizations of the equivalence of disjunctive aggregate queries, which generalize our previous results for the conjunctive case.

For each operator α, we introduce several types of queries using views as candidates for rewritings. We unfold such a candidate by replacing each occurrence of a view predicate with its definition, thus obtaining a regular aggregate query. The candidates have a different, usually more complex operator than α. We prove that unfolding the candidate, however, results in a regular aggregate query that is equivalent to the candidate modulo the view definitions. This property justifies considering these types of queries as *natural* candidates for rewritings. In this way, we reduce the problem of whether there exist rewritings of a particular type to a problem involving equivalence.

We distinguish between partial rewritings that contain at least one view predicate and complete rewritings that contain only view predicates. In contrast to previous work on this topic, we not only give sufficient, but also necessary conditions for a rewriting to exist. More precisely, we show for each type of candidate that the existence of both, partial and complete rewritings is decidable, and we provide upper and lower complexity bounds.

1 Introduction

Rewriting queries using views is a fundamental problem in databases, which has attracted considerable attention. View usability techniques have applications in a number of areas. In query optimization, the execution of a query can be accelerated if results from previous queries can be used to compute answers [YL87, CR94, CKPS95]. In designing information systems over which a huge number of a priori known queries are posed periodically, it can be beneficial to store such intermediate results beforehand that are useful for as many queries as possible [LFS97, RSS96]. Integrating heterogeneous information sources is another problem which may be reduced to the view usability problem [LSK95].

While the focus of this work was for a long time on queries without aggregation, interest in aggregate queries has been motivated recently by the surge of data warehousing and decision support applications, where queries of this kind typically occur. Optimization based on the reuse of previously computed results is particularly promising for aggregate queries, since often huge numbers of data items are processed to produce a single aggregate value. In fact, most existing data warehouses make use of this idea in their optimization algorithms in a more or less ad hoc way [Kim96].

In this paper we create a framework for studying the view usability problem for aggregate queries. The queries and views we consider correspond to unnested SQL queries, possibly with union, that employ the operators *min*, *max*, *count*, and *sum*. In contrast to previous work on this topic, we not only give sufficient, but also necessary conditions for a rewriting of a certain type to exist.

One contribution of this paper are syntactic characterizations of the equivalence of disjunctive aggregate queries, which generalize our previous results for the conjunctive case [NSS98]. Another contribution is our "unfolding technique" by which we reduce the problem of view usability to a problem involving equivalence. The characterizations of equivalences of disjunctive queries thus enable us to extend our view rewriting results to the disjunctive case.

For each aggregate operator, we introduce several types of queries using views as candidates for rewritings. We distinguish between *partial* rewritings that contain *at least one* view predicate and *complete* rewritings that contain *only* view predicates. We show for each type of candidate that the existence of both partial and complete rewritings is decidable, and we provide upper and lower complexity bounds.

In Section 2, we collect basic definitions for non-aggregate queries. In Section 3 we extend the definitions to queries *with* aggregates. In Section 4 we give syntactic characterizations of the equivalence of *disjunctive* aggregate queries. The core of the paper is Section 5, where we develop a theory of rewriting aggregate queries using views with and without aggregates. Section 6 illustrates our rewriting techniques with examples. In Section 7, we study the complexity of recognizing and finding rewritings. In Section 8, we survey related work and conclude.

2 Preliminaries

We introduce conjunctive and disjunctive queries and review their basic properties. We use standard Datalog syntax extended by aggregate functions. The goal of this section is to present basic definitions that will be necessary to provide a framework for analyzing unnested aggregate queries as they are definable in SQL without using the `having` construct.

2.1 Syntax of Disjunctive Queries

We assume that there is an infinite set of predicate symbols, which are denoted as p, q, r. We denote the database as $\mathcal{D}$. A *term,* denoted as s, t, is either a variable or a constant. A *relational atom* has the form $p(s_1, \ldots, s_k)$, where p is a predicate of arity k. We also use the notation $p(\bar{s})$, where $\bar{s}$ stands for a tuple of terms $(s_1, \ldots, s_k)$. Similarly, $\bar{x}$ stands for a tuple of variables. An *ordering atom* or *comparison* has the form $s_1\ \rho\ s_2$, where ρ is one of the ordering predicates $<$, $\leq$, $>$, or $\geq$. An *atom* is a relational atom or a comparison. A *condition,* denoted as A, is a conjunction of atoms. We assume that conditions are safe [Ull88].

A *query* is a non-recursive expression of the form

$$q(\bar{s}) \leftarrow A_1 \vee \ldots \vee A_n,$$

where each A_i is a condition containing all the variables appearing in the tuple $\bar{s}$.

A query is *conjunctive* if it contains only one disjunct. A query is *linear* if no disjunct contains two relational atoms with the same predicate symbol. A query is *relational* if it contains only relational atoms (i.e. it does not contain comparisons). By abuse of notation, we will often refer to a query by its head $q(\bar{s})$ or simply by the predicate of its head q.

2.2 Semantics of Disjunctive Queries

We define in which way a query q, evaluated over a database $\mathcal{D}$, gives rise to a *set* of tuples $q^{\mathcal{D}}$ or a *bag,* that is a multiset, of tuples $\{\!\{q\}\!\}^{\mathcal{D}}$.

An *assignment* γ for a condition A is a mapping of the variables appearing in A to constants, and of the constants appearing in A to themselves. Assignments are naturally extended to tuples and atoms. *Satisfaction* of atoms and of conjunctions of atoms by an assignment w.r.t. a database are defined in the obvious way. For $\bar{s} = (s_1, \ldots, s_k)$ we let $\gamma\bar{s}$ denote the tuple $(\gamma(s_1), \ldots, \gamma(s_k))$.

Under *set semantics,* a query $q(\bar{s}) \leftarrow A_1 \vee \ldots \vee A_n$ defines a new relation $q^{\mathcal{D}}$, for a given database $\mathcal{D}$, as follows:

$$q^{\mathcal{D}} := \bigcup_{i=1}^{n} \{\gamma\bar{s} \mid \gamma \text{ satisfies } A_i \text{ w.r.t. } \mathcal{D}\}.$$

Bag-set semantics has been introduced by Chaudhuri and Vardi [CV93] to semantically model query execution by SQL-based database systems. There, the database contains relations, i.e. sets of tuples, while a query returns a bag, i.e. a multset of tuples. We denote the union of two multisets M_1, M_2 as $M_1 \uplus M_2$. The multiplicity with which an element occurs in the union is the sum of the multiplicities with which it occurs in the single multisets. We formally define $\{\!\{q\}\!\}^{\mathcal{D}}$ in analogy to $q^{\mathcal{D}}$ as:

$$\{\!\{q\}\!\}^{\mathcal{D}} := \biguplus_{i=1}^{n} \{\!\{\gamma\bar{s} \mid \gamma \text{ satisfies } A_i \text{ w.r.t. } \mathcal{D}\}\!\}.$$

The definition means that for each condition A_i we collect all satisfying assignments γ. This collection is a set. Then we apply the assignments γ to the output variables $\bar{s}$. Since $\gamma\bar{s}$ may be the same tuple for different assignments γ, the result is a multiset. Finally, we take the multiset union of all the multisets obtained in this way.

Also *bag semantics* has been introduced by Chaudhuri and Vardi [CV93]. The only difference between bag semantics and bag-set semantics is that in the first case the database contains multisets of tuples while in the second case it contains sets of tuples.

Two queries q and q' are *equivalent* under set-semantics, or *set-equivalent*, if over every database they return the same sets of results. Similarly, q and q' are *equivalent* under bag-set-semantics, or *bag-set-equivalent*, if over every database they return the same multisets of results, that is, $\{\!\{q\}\!\}^{\mathcal{D}} = \{\!\{q'\}\!\}^{\mathcal{D}}$ for all databases $\mathcal{D}$. Under set semantics, equivalence of conjunctive and disjunctive queries can be decided by checking whether there exist *containment mappings* or *homomorphisms* between the queries [CM77, JK83, SY81].

3 Aggregate Queries

In this section we extend the general framework presented in Section 2 to aggregate queries. Such queries are evaluated in two phases. In the first phase, the query retrieves a multiset of tuples from the database. The tuples are then grouped into equivalence classes, and to each equivalence class an aggregation function is applied. We define first aggregation functions and then introduce the syntax and semantics of aggregate queries.

In [NSS98] we showed that equivalence of conjunctive queries with a number of aggregate terms can be easily reduced to equivalence of queries with a single aggregate term. This can be generalized for disjunctive queries. Also, rewritings of queries with several aggregate terms can be constructed from rewritings with a single term. Thus, in this paper we consider only queries having a single aggregate term in the head.

3.1 Aggregation Functions

We assume in this paper that the data we want to aggregate are real numbers. We denote the set of real numbers as $\mathbf{R}$. If S is a set we denote by $\mathcal{M}(S)$ the set of finite multisets over S. A *k-ary aggregation function* is a function $\alpha\colon \mathcal{M}(\mathbf{R}^k) \rightarrow \mathbf{R}$ that maps multisets of k-tuples of real numbers to real numbers. The aggregate queries that we consider in this paper have the aggregation functions *count*, *sum*, and *max*. Results for queries with the function *min* are analogous to results for *max*-queries. Therefore, we do not consider *min*. Note that our function *count* is analogous to the *count*$(*)$ function of SQL.

An *aggregate term* is an expression built up using variables, the operations addition and multiplication, and aggregate functions.[1] For example *count* and $sum(z_1 * z_2)$, are aggregate terms. Every aggregate term gives rise to an aggregation function in a natural way. We use κ and λ as abstract notations for aggregate terms. If we want to refer to the variables occurring in an aggregate term, we write $\kappa(\bar{y})$ and $\lambda(\bar{y})$, where $\bar{y}$ is a tuple of distinct variables, consisting of the variables in κ or λ, respectively.

We call terms of the form *count*, $sum(y)$ and $max(z)$ *elementary aggregate terms.* Abstractly, elementary aggregate terms are denoted as $\alpha(y)$, where α is an aggregate function. We call queries with elementary aggregate terms *elementary queries.*

In this paper we are interested in equivalences of elementary queries and in rewriting elementary queries using elementary views.

3.2 Syntax of Aggregate Queries

In this subsection we define formally the syntax of aggregate queries. An *aggregate query* is a query augmented by an aggregate term in its head. Thus it has the form

$$q(\bar{s}, \kappa(\bar{y})) \leftarrow A_1 \vee \ldots \vee A_n. \tag{1}$$

In addition, we require that

- $\kappa(\bar{y})$ is an aggregate term;
- no variable $x \in \bar{s}$ occurs in $\bar{y}$;
- each condition A_i contains all the variables in $\bar{s}$ and in $\bar{y}$.

We call $\bar{s}$ the *grouping terms* of the query. If the disjuncts are not important we write B as an abstract notation for the body of a query. 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).

3.3 Semantics of Aggregate Queries

Consider an aggregate query q as in Equation (1). 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

$$\breve{q}(\bar{s}, \bar{y}) \leftarrow A_1 \vee \ldots \vee A_n, \tag{2}$$

The core is the query that returns all the values that are amalgamated in the aggregate.

Then, we construct multisets as arguments for the aggregate function $\kappa(\bar{y})$ out of the assignments satisfying the core. Let $,_i$ be the set of assignments to the variables in the condition A_i that satisfy A_i, and let $, := \biguplus_{i=1}^{n} ,_i$ be the multiset union of the $,_i$.

For a tuple $\bar{d}$, let

$$,_{\bar{d}} \quad := \quad \{\!\{ \gamma \in , \mid \gamma(\bar{s}) = \bar{d} \}\!\}.$$

In the bags $,_{\bar{d}}$, we group those satisfying assignments that agree on $\bar{s}$. Therefore, we call $,_{\bar{d}}$ the *group* of $\bar{d}$. The tuple $\bar{y}$ contains the argument variables of the aggregate term in the head of q. Then

$$,_{\bar{d}}(\bar{y}) \quad := \quad \{\!\{ \gamma(\bar{y}) \mid \gamma \in ,_{\bar{d}} \}\!\}$$

is the multiset of tuples obtained by restricting assignments in $,_{\bar{d}}$ to $\bar{y}$. Now we define the result of evaluating q over $\mathcal{D}$ as

$$q^{\mathcal{D}} \quad := \quad \{(\bar{d}, e) \mid \bar{d} = \gamma(\bar{s}) \text{ for some } \gamma \in , , \text{ and } e = \kappa(,_{\bar{d}}(\bar{y}))\}.$$

where we interpret the aggregate term κ as an aggregation function.

4 Equivalence of Aggregate Queries

This section contains characterizations of equivalences of aggregate queries. They generalize earlier results for conjunctive aggregate queries [NSS98]. The characterizations of equivalences of disjunctive queries enable us to extend our view rewriting results to the disjunctive case. Formally, two aggregate queries q, q' are *equivalent* if for every database $\mathcal{D}$ they define the same relation, that is, $q^{\mathcal{D}} = q'^{\mathcal{D}}$.

Our result about disjunctive *count*-queries shows that equivalence of such queries under bag-set-semantics is decidable, and a slightly changed characterization shows that the same is true for bag-semantics. This is remarkable, since Ioannidis and Ramakrishnan [IR95] proved that containment of disjunctive queries under bag-semantics is undecidable.

4.1 Reduced Queries

Our characterizations of equivalence of *count* and *sum*-queries rely on a specific normal form of conjunctive queries. We say that a conjunctive query $q(\bar{s}) \leftarrow R \;\&\; C$ is *reduced* if

- there are no variables x, y occurring in C such that $C \models x = y$;
- there is no variable x occurring in C such that $C \models x = d$ for a constant d.

We have shown that for every conjunctive query one can compute in polynomial time an equivalent reduced conjunctive query [NSS98].

4.2 Linear Expansion

In general, the comparisons in the body of a query induce a partial order among the terms of the query. Such a partial order contains disjunctive information, since it does not specify completely how the terms are related to each other. To deal with equivalence of arbitrary queries, which may have comparisons, we have to consider all linear orders to which such a partial order can be extended. We describe now how to create such linearizations.

Let $T = D \cup W$ be a set of terms, where D is a set of constants and W is a set of variables. A *linearization* of T is a set of comparisons L over the terms in T, such that for any s, $t \in T$, the set L implies exactly one of $s < t$, $s = t$, or $s > t$.

Thus, a linearization L partitions the terms into equivalence classes, such that the terms in each class are equal and the classes are arranged in a strict linear order. In each class of L, there is at most one constant. Otherwise, L would be unsatisfiable and entail any consequence.

We consider the conjunctive query $q(\bar{s}) \leftarrow R \;\&\; C$ where R is a conjunction of relational atoms and C is a conjunction of comparisons. Let D be a set of constants including those appearing in q and let W be the set of variables occurring in q. Let L be a linearization of $D \cup W$ that is compatible

[1]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.

with the comparisons in C, that is, $L \cup C$ is satisfiable. Then an equivalent reduced version of $q(\bar{t}) \leftarrow R \,\&\, L$ is called a *linearization* of q w.r.t. L.

Example 4.1 Consider the query

$$q(x_1, x_2) \leftarrow p(x_1, x_2) \,\&\, 0 \leq x_1 \,\&\, 0 < x_2.$$

Then the following are linearizations of $\{0, x_1, x_2\}$ that are compatible with the comparisons in q:

$$\begin{aligned} L_1 &= \{0 = x_1 \,\&\, x_1 < x_2\} \\ L_2 &= \{0 < x_1 \,\&\, x_1 < x_2\} \\ L_3 &= \{0 < x_1 \,\&\, x_1 = x_2\}. \end{aligned}$$

Conjoining them with q gives rise to the reduced queries

$$\begin{aligned} q_1(0, x_2) &\leftarrow p(0, x_2) \,\&\, 0 < x_2 \\ q_2(x_1, x_2) &\leftarrow p(x_1, x_2) \,\&\, 0 < x_1 \,\&\, x_1 < x_2 \\ q_3(x_1, x_1) &\leftarrow p(x_1, x_1) \,\&\, 0 < x_1, \end{aligned}$$

which are linearizations of q.

To a term linearization L, there may corresond more than one linearization of q. For instance, conjoining q with L_3 and reducing it can also produce the query $q_3'(x_2, x_2) \leftarrow p(x_2, x_2) \,\&\, 0 < x_2$. It is easy to see that all reduced versions of a query are isomorphic, that is, they are the same up to a renaming of variables.

If D contains the constants of q and W is the set of variables of q, we denote by $\mathcal{L}_D(q)$ the set of linearizations of $D \cup W$ that are compatible with the comparisons of q.

If $q(\bar{s}) \leftarrow A$ is a conjunctive query, then a *linear expansion* of q is a union of queries

$$q^{lin} = \bigvee_{L \in \mathcal{L}_D(q)} q_L,$$

where each q_L is a linearization of q w.r.t. L. The semantics of unions of queries is defined as one would expect. That is, under set semantics, a union of queries returns the union of the set of tuples that are the results of the single queries in the union. Similarly, under bag and bag-set-semantics, it returns a multiset union. Note, that the linear expansion of a query differs depending on whether the comparisons are interpreted over the integers or over the rational numbers, since more sets of comparisons are satisfiable over the rationals than over the integers.

Let $q(\bar{s}) \leftarrow A_1 \vee \ldots \vee A_n$ be a disjunctive query. For $i = 1, \ldots, n$ we define conjunctive queries $q_i(\bar{s}) \leftarrow A_i$. Then a linear expansion of q is a union of linear expansions of the q_i, that is, a query of the form

$$q^{lin} = \bigvee_{i=1}^{n} \bigvee_{L \in \mathcal{L}_{D_i}(q_i)} q_{iL}.$$

If q and the D_i are clear from the context, we simply write $q^{lin} = \bigvee_{i \in I} \bigvee_L q_{iL}$.

Proposition 4.2 *Let q be a disjunctive query and q^{lin} be a linear expansion of q. Then q and q^{lin} are equivalent, both under set-semantics and under bag-set-semantics.*

[2]We denote as GI the class of problems that are many-one-reducible to the graph isomorphism problem.

4.3 Count-Queries and Bag-Set-Equivalence

Two *count*-queries are equivalent if they return the same results with the same multiplicities. This means that, under bag-set-semantics, they return the same multisets. Therefore equivalence of *count*-queries is the same as equivalence of their cores under bag-set-semantics. Thus, in the following, we investigate bag-set-equivalence of disjunctive non-aggregate queries.

We say that two unions of queries $q = \bigvee_{i \in I} q_i$ and $q' = \bigvee_{j \in J} q_j'$ are *isomorphic* if there is a bijective mapping $\mu: I \to J$ such that q_i and $q_{\mu(i)}'$ are isomorphic for all $i \in I$.

Theorem 4.3 (Isomorphism Implies Bag-Set-Equivalence) *Let q and q' be two disjunctive queries. If q and q' have isomorphic linear expansions, then they are bag-set-equivalent.*

In order to prove the converse of the preceding theorem, we have to control the set of constants over which we take the linear expansion.

Theorem 4.4 (Bag-Set-Equivalence Implies Isomorphism) *Let $q(\bar{s})$ and $q'(\bar{s}')$ be two disjunctive queries, and let D be the set of constants occurring in q or q'. Let q^{lin} be a linear expansion of q over D and ${q'}^{lin}$ be an analogous linear expansion of q' over D. If q and q' are bag-set-equivalent, then q^{lin} and ${q'}^{lin}$ are isomorphic.*

For the relational case we have shown a simpler characterization.

Theorem 4.5 (Relational Queries) *Two relational queries are bag-set-equivalent if and only if they are isomorphic.*

The theorems of this subsection also hold verbatim for bag-equivalence of queries if in the definition of isomorphism we take into account the multiplicity of relational atoms in a query (see also [CV93]).

4.4 Equivalence of Sum-Queries

We have extended our characterization for equivalence of conjunctive *sum*-queries to the disjunctive case. This extension is analogous to our extension for *count*-queries and is not presented due to lack of space. The characterization shows that deciding the equivalence of disjunctive *sum*-queries is in PSPACE. Thus it is interesting to note that deciding equivalence of disjunctive *sum*-queries is no more difficult than deciding equivalence of conjunctive *sum*-queries.

We have also shown that equivalence of *sum*-queries that without constants or without comparisons can be reduced to bag-set-equivalence.

Theorem 4.6 (Equivalence of Sum-Queries) *Let q and q' be sum-queries without constants or without comparisons and $\breve{q}$ and $\breve{q}'$ be their cores. Then the following are equivalent:*

1. *q and q' are equivalent;*
2. *$\breve{q}$ and $\breve{q}'$ are bag-set-equivalent;*
3. *$\breve{q}$ and $\breve{q}'$ are isomorphic.*

aggregate function	q arbitrary q' arbitrary	q relational q' arbitrary	q relational q' relational	q linear q' arbitrary	q linear q' linear
count, *sum*	GI-hard[2]/in PSPACE	GI-complete	GI-complete	polyn.	polyn.
max	Π_2^P-complete	Π_2^P-complete	NP-complete	NP-hard/in Π_2^P	polyn.

Table 1: Complexity of Equivalence

4.5 Equivalence of Max-Queries

Equivalence of *max*-queries can be reduced to a property of non-aggregate queries that we call dominance. Let $q(\bar{s}, t)$ and $q'(\bar{s}', t')$ be two queries. We say that q *is dominated* by q' if for every database, whenever q returns a tuple $(\bar{d}, d)$, then q' returns a tuple $(\bar{d}, d')$ with $d' \geq d$. We say that q and q' *dominate each other* if q is dominated by q' and q' is dominated by q.

Proposition 4.7 *Two disjunctive* max*-queries are equivalent if and only if their cores dominate each other.*

Dominance of disjunctive queries is Π_2^P-complete. It can be checked in a fashion similar to checking containment.

Obviously, if a query q is contained in another query q', then q is dominated by q'. If the queries are relational, then the converse also holds.

Theorem 4.8 (Dominance is Containment) *Consider relational disjunctive queries q and q'. Then q is dominated by q' if and only if q is contained in q'.*

4.6 Complexity of Equivalence

Our syntactic characterizations of equivalences of disjunctive aggregate queries can be translated into algorithms and thus give rise to upper bounds for the complexity of deciding equivalence. The lower bounds hold already for conjunctive aggregate queries (see [NSS98]). We denote as GI the class of problems that are many-one-reducible to the graph isomorphism problem.

Theorem 4.9 (Disjunctive Queries)

- *Checking the equivalence of disjunctive* sum *or* count*-queries is GI-hard and in* PSPACE.
- *Checking the equivalence of disjunctive* max*-queries is Π_2^P-complete.*

Our characterizations of special cases give rise to more specific complexity bounds. In addition, we have refined our characterizations to cover also asymmetric cases, where one query is relational or linear and the second one is arbitrary. Lower bounds for equivalences among conjunctive aggregate queries (see [NSS98]) yield also lower bounds for disjunctive queries. We also found asymmetric cases where the lower bounds differ from the lower bounds for more specific symmetric cases.

We summarize our results about the complexity of equivalence checking for disjunctive aggregate queries in Table 1. For conjunctive queries the table would be exactly the same. Thus, it is remarkable that deciding equivalences among disjunctive queries is no more difficult than deciding equivalences among conjunctive queries.

5 Rewriting of Aggregate Queries

Given a set of queries $\mathcal{V}$ called "views" and an aggregate query q, our goal is to find a new query r, that uses data base relations and some views, such that the evaluation of q and the evaluation of r yield the same result over all databases. In our set of views we allow both aggregate and non-aggregate queries. We can assume that for each aggregate query we have an analogous non-aggregate query obtained by projecting out the aggregate term. We assume, w.l.o.g., throughout this paper that the sets of variables used in the views are disjoint.

5.1 Equivalence Modulo a Set of Views

The relationship between a query q and its rewriting r is not simply equivalence of queries, because the views are not additional 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 $\mathcal{D}$.

5.2 Rewritings

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 $q \equiv_\mathcal{V} r$. If r is a rewriting of q using $\mathcal{V}$, we say that r is a *partial rewriting of* q *using* $\mathcal{V}$ if r contains at least one atom with a predicate from $\mathcal{V}$ and that r is a *complete rewriting of* q *using* $\mathcal{V}$ if r contains only atoms with predicates from $\mathcal{V}$.

For simplicity of exposition, we will consider only complete rewritings. Any definitions and characterizations proposed for complete rewritings are applicable also for partial ones in a very intuitive way, i.e. by extending the set of views by queries that define the atoms in the original query. However, searching for partial rewritings may give rise to simpler algorithms than searching for complete rewritings.

Thus, we will only differentiate between complete and partial rewritings when considering complexity results.

Let r be a query, aggregate or non-aggregate, and a be a relational atom in the body of r. A query r' is a *diminution of r by a* if r' is obtained from r by dropping a from the body of r and possibly adding comparisons to r. We say that r is *diminishable modulo $\mathcal{V}$* if there is an atom a such that r and a diminution by a are equivalent modulo $\mathcal{V}$. Otherwise, r is *undiminishable.*

Undiminishable queries are in general preferable to diminishable queries because fewer relations have to be accessed to evaluate them. For this reason, we are interested in undiminishable rewritings.

5.3 Conjunctive Rewritings

In this subsection we consider conjunctive rewritings, i.e. rewritings that are of a conjunctive form and use only conjunctive views. This simplifies the presentation. We relax this restriction in the sequel.

Compared to the rewriting problem for non-aggregate conjunctive queries, there is an additional complication when we are looking for rewritings of aggregate queries. In the case of non-aggregate conjunctive queries, the candidates for rewritings are conjunctive queries over the views. In the case of aggregate queries, even if we admit only queries whose body is a conjunction of atoms, there is an infinity of possible aggregate functions that one could use in a rewriting.

Example 5.1 Consider the three aggregate queries

$$\begin{aligned} q_1(x, count) &\leftarrow p(x, y_1) \,\&\, p(x, y_2) \\ v_1(x, count) &\leftarrow p(x, y) \\ r_1(x, z^2) &\leftarrow v_1(x, z). \end{aligned}$$

Then it is easy to see that r_1 is a rewriting of q_1 modulo $\{v_1\}$.

The aggregate function in a rewriting need not be a polynomial, but can also be a root. Consider

$$\begin{aligned} q_2(x, count) &\leftarrow p(x, y) \\ v_2(x, count) &\leftarrow p(x, y_1) \,\&\, p(x, y_2) \\ r_2(x, \sqrt{z}) &\leftarrow v_2(x, z). \end{aligned}$$

Again, r_2 is a rewriting of q_2 modulo $\{v_2\}$.

This example shows that it may be very hard to make statements about the existence of rewritings and to search for them if we do not restrict the class of candidates to be considered. Our next step is therefore to identify for each aggregate operator a class of queries that we consider as natural candidates for rewritings of queries with that operator.

To illustrate the definitions and the process of rewriting, we will give lifelike examples in Section 6.

5.3.1 Rewriting Candidates

For each aggregate operator, we restrict the kind of views that we want to consider for rewritings. We call those views *valid* views. First, a view with the same aggregate operator as the query is valid. For operators that are sensitive to the multiplicity of tuples, namely *count* and *sum*, we admit as additional views only *count* views. For the operator *max*, which is insensitive to multiplicities, we consider non-aggregate views. As mentioned before, this includes also views that are obtained from aggregate views by strippin off their aggregate argument.

Definition 5.2 (Valid Views) *Let v be a view and α an aggregate function. We say that v is a* valid view *for rewriting an α-query q if one of the following holds:*

- *v is an α-query (e.g., v and q are both sum-queries);*
- *v is a non-aggregate query and α is the function max;*
- *v is a count-query and α is the function sum.*

In the sequel we will assume that all views used in rewritings are valid. In Table 2 we summarize which kinds of views are valid for which kind of aggregate query. Since for every aggregate query there is an analogous non-aggregate query obtained by projecting out the aggregate term, we allow in fact arbitrary aggregate views in *max*-query rewritings.

Aggregation Function	Valid Views
count	*count*
sum	*count*, *sum*
max	*max*, non-aggregate

Table 2: Views valid for aggregation functions

We call *sum* and *max*-queries *parametric queries,* and *count* and non-aggregate queries *non-parametric queries.* In a similar way, we talk about parametric and non-parametric views. Parametric queries and views have an aggregate operator that takes an argument.

Now, for each of the aggregate operators *sum* and *max* we are going to define two classes of queries that we consider as natural candidates for rewritings. Theorem 5.10 will justify our choice.

First, we specify the body of the candidates. Relational atoms in the body that are formed with the predicate of a non-aggregate view are allowed to be arbitrarily instantiated. Atoms that are formed with the predicate of an aggregate view, say the view $v(\bar{s}, \kappa)$, are only allowed to have the form $v(\theta\bar{s}, z)$. This means, the non-aggregate arguments are instantiated, but the aggregate argument is not. We will write θv as a shorthand notation for an atom with the predicate v if the arguments $\bar{s}$ and z are either clear or unimportant. Next, we provide a technical definition.

Definition 5.3 (Output Variable) *Let $v(\bar{s}, \kappa)$ be an aggregate view and $v(\theta\bar{s}, z)$ an instantiation of v. Then z is the* output variable *of the atom $v(\theta\bar{s}, z)$. If the atom is clear from the context, we will simply say that z is the output variable of v.*

The candidates for rewriting queries with an operator $\alpha \in \{sum, max\}$ have the general form

$$r(\bar{s}, \alpha(y * \prod \bar{z})) \quad \leftarrow \quad B, \tag{3}$$

where all views occurring in B are valid for α, and y and $\bar{z}$ are output variables of aggregate views occurring in B. Moreover, each output variable occurs only once in B, and no output variable is involved in a comparison. The last condition ensures that aggregates in a view are not subjected to selection conditions.

Note that the expression $y * \prod \bar{z}$ is a shorthand notation for the product of all the variables $z_i \in \bar{z}$ and y. Note that by the constraints given below, the tuple $\bar{z}$ will always be empty for candidates for rewriting *max*-queries and thus the product of $\bar{z}$ will be 1. The notation was introduced thus for uniformity.

For each operator α we distinguish between those candidates that use a view with α (called α-view candidates) and those that do not (called pure α-candidates).

Definition 5.4 (α-view Candidates) *The query defined by Equation (3) is an* α-view candidate *if the following hold:*

- *B has exactly one occurrence of an α-view, v, and y appears as the output variable of v;*
- *$z \in \bar{z}$ if and only if z appears as an output variable of a non-α-view.*

Definition 5.5 (Pure α-Candidates) *The query defined by Equation (3) is a* pure α-candidate *if the following hold:*

- *B has no occurrences of α-views, and y appears in B;*
- *$z \in \bar{z}$ if and only if z appears as an output variable of a view.*

We extend the definition of *pure α-candidates* to *count*-views and define a pure *count*-candidate as a query of the form

$$r(\bar{s}, sum(\prod \bar{z})) \leftarrow B,$$

where $z \in \bar{z}$ if and only if z appears only as an output variable of a view.

Let α be one of the aggregation functions *max*, *sum* and let $\kappa = \alpha(y)$. Then we write $f^{\kappa}(\prod \bar{z})$ as a shorthand for $\alpha(y * \prod \bar{z})$. If α is *count* and $\kappa = count$, then $f^{\kappa}(\prod \bar{z})$ is a shorthand for $\alpha(y * \prod \bar{z})$. We say that κ is the *canonical aggregate term* for α.

Definition 5.6 (Rewriting Candidates) *A query r is an* α-rewriting candidate *if it is an α-view candidate or a pure α-candidate. For α-rewriting candidates we employ the generic notation*

$$r(\bar{s}, f^{\kappa}(\prod \bar{z})) \leftarrow B,$$

where κ is the canonical aggregate term for α.

To make our generic definition more intuitive, we specify in Table 3 the format of the candidates for the different aggregate functions. Recall that for *count* only pure candidates are defined.

For clarity we use the notation v^c to denote *count*-views. Similarly, v^m and v^s denote *max* and *sum*-views, respectively, while v (without a superscript) denotes a non-aggregate view. The symbol C stands for a conjunction of comparisons.

5.3.2 Unfoldings of Candidates

In this section we reduce the problem of verifying whether a rewriting candidate is in fact a rewriting of a given query to the problem of checking equivalence of aggregate queries, which was dealt with in Section 4.

Essentially, in order to check whether a rewriting candidate r is in fact a rewriting of an aggregate query q, we unfold the view predicates in the body of r, thus obtaining a condition that consists only of base predicates. The aggregate query whose body is this unfolded condition and whose head has the same arguments as r is denoted as r^u. Then r is a rewriting of q if and only if r^u and q are equivalent.

In order to properly define the unfolding, we have to specify how to instantiate the bodies of the views used in the candidate. Special care has to be taken for the aggregation variable.

Definition 5.7 (Extension) *Let $v(\bar{s}, \alpha(y))$ be a parametric view and $\theta v = v(\theta\bar{s}, z)$ be an instantiation of v. Then the* extension *of θ for θv is $\hat{\theta} := \theta \cup \{z/y\}$. If v is non-parametric, then the* extension *of θ for θv is $\hat{\theta} := \theta$.*

Extensions will be used to instantiate the bodies of views in an unfolding. They are defined in such a way that an extension replaces the output variable of a view with the parameter of the original aggregate function.

Definition 5.8 (Unfolding) *Let $r(\bar{s}, f^{\kappa}(\prod \bar{z})) \leftarrow B$ be a rewriting candidate. Then the* unfolding *of r is the query*

$$r^u(\bar{s}, \kappa) \leftarrow B',$$

where B' has been obtained from B by replacing each view atom θv with the condition $\hat{\theta} B_v$ that is obtained by instantiating the body B_v of v with the extension of θ for θv.

The unfolding operation mimics the two phase evaluation of the original query r over a database $\mathcal{D}_{\mathcal{V}}$, as defined in Subsection 5.1. Thus the following theorem holds.

Theorem 5.9 (Unfolding Preserves Equivalence) *Let r be a rewriting candidate of a query over $\mathcal{V}$ and let r^u be its unfolding. Then $r \equiv_{\mathcal{V}} r^u$.*

Proof. (Sketch) The proof is by a case analysis according to the different rewriting candidates and aggregate functions. For the sake of simplicity we present in this paper only the proof for *count*-rewriting candidates. Recall that for *count* only pure rewriting candidates are defined (see Table 3).

A *count*-rewriting candidate has the form:

$$r(\bar{s}, sum(\prod_{i=1}^{n} z_i)) \leftarrow v_1(\theta_1 \bar{s}_1, z_1) \;\&\; \ldots \;\&\; v_n(\theta_n \bar{s}_n, z_i) \;\&\; C.$$

As we have shown in the long version of this paper, one can assume that the views in r are not instantiated. Therefore we assume w.l.o.g. that r has the form

$$r(\bar{s}, sum(\prod_{i=1}^{n} z_i)) \leftarrow v_1(\bar{s}_1, z_1) \;\&\; \ldots \;\&\; sv_n(\bar{s}_n, z_i) \;\&\; C,$$

with the unfolding $r^u(\bar{s}, count) \leftarrow B_1 \;\&\; \ldots B_n \;\&\; C$.

Let $\mathcal{D}$ be a database, $\bar{d} \in |\mathcal{D}|^n$, and d be a natural number. We show that $(\bar{d}, d) \in r^{\mathcal{D}_{\mathcal{V}}}$ if and only if $(\bar{d}, d) \in (r^u)^{\mathcal{D}}$.

"Only if." Suppose that $(\bar{d}, d) \in r^{\mathcal{D}_{\mathcal{V}}}$. Then there are assignments $\phi_1, \ldots, \phi_m$ such that

1. ϕ_j satisfies the body of r over $\mathcal{D}_{\mathcal{V}}$;
2. $\phi_j(\bar{s}) = \bar{d}$, for all $j \in 1..m$;
3. $d = \sum_{j=1}^{m} \prod_{i=1}^{n} \phi_j(z_i)$.

For each i, j, we define the set Ψ_{ij} as consisting of assignments ψ that are defined for the variables occurring in the body B_i of v_i such that ψ satisfies B_i and $\psi(\bar{s}_i) = \phi_j(\bar{s}_i)$. That is, Ψ_{ij} contains those assignments that are counted by v_i as producing the output $\phi_j(\bar{s}_i)$ for v_i. Thus, the set Ψ_{ij} has $\phi_j(z_i)$ elements. Let $\psi_1, \ldots, \psi_n$ be assignments such that for some $j \in 1..m$ it holds that $\psi_i \in \Psi_{ij}$. We show that each assignment ϕ^u that satisfies the body of r^u and maps $\bar{s}$ to $\bar{d}$ can be constructed in a unique way from such ψ_i, and that, conversely, each such ϕ^u gives rise to a unique sequence

Aggregation Function	Candidate Type	Rewriting Candidate
count	pure	$r(\bar{s}, sum(\prod_{i=1}^{n} z_i)) \leftarrow v_1^c(\theta_1\bar{s}_1, z_1)\ \&\ \ldots\ \&\ v_n^c(\theta_n\bar{s}_n, z_n)\ \&\ C$
sum	pure	$r(\bar{s}, sum(y * \prod_{i=1}^{n} z_i)) \leftarrow v_1^c(\theta_1\bar{s}_1, z_1)\ \&\ \ldots\ \&\ v_n^c(\theta_n\bar{s}_n, z_n)\ \&\ C$
	sum-view	$r(\bar{s}, sum(y * \prod_{i=1}^{n} z_i)) \leftarrow v^s(\theta\bar{s}_s, y)\ \&\ v_1^c(\theta_1\bar{s}_1, z_1)\ \&\ \ldots\ \&\ v_n^c(\theta_n\bar{s}_n, z_n)\ \&\ C$
max	pure	$r(\bar{s}, max(y)) \leftarrow v_1(\theta_1\bar{s}_1)\ \&\ \ldots\ \&\ v_n(\theta_n\bar{s}_n)\ \&\ C$
	max-view	$r(\bar{s}, max(y)) \leftarrow v^m(\theta\bar{s}_m, y)\ \&\ v_1(\theta_1\bar{s}_1)\ \&\ \ldots\ \&\ v_n(\theta_n\bar{s}_n)\ \&\ C$

Table 3: Patterns of Rewriting Candidates

of ψ_i's. Since each Ψ_{ij} has $\phi_j(z_i)$ elements, the number of such assignments ϕ^u is $d = \sum_{j=1}^{m} \prod_{i=1}^{n} \phi_j(z_i)$. From this, we conclude that the query r^u returns the tuple $(\bar{d}, d)$ over $\mathcal{D}$.

"If." Suppose that r^u returns $(\bar{d}, d')$ over $\mathcal{D}$. Then there is at least one mapping ϕ^u such that $\phi^u(\bar{s}) = \bar{d}$ and ϕ^u satisfies the body of r^u. As before, we construct assignments $\psi_1, \ldots, \psi_n$ out of ϕ^u, and an assignment ϕ to the variables of r out of the ψ_i. By construction, ϕ satisfies the body of r and $\phi(\bar{s}) = \bar{d}$. Hence, there is a number d such that $(\bar{d}, d) \in r^{\mathcal{D}}$. Therefore, as we have shown above, $(\bar{d}, d) \in (r^u)^{\mathcal{D}}$. However, r^u is an aggregate query and aggregates depend functionally on their group. Thus, $d = d'$ and $(\bar{d}, d') \in r^{\mathcal{D}}$. □

Our choice of aggregate functions, $f^\kappa(\prod \bar{z})$, may seem arbitrary at first glance. However, if we expect the query obtained by unfolding the rewriting to be equivalent to the rewriting modulo $\mathcal{V}$, then this function is the only choice. We make this more precise in the following theorem.

Theorem 5.10 (Natural Rewriting Candidates) *Let $r(\bar{s}, f^\kappa(\prod \bar{z})) \leftarrow B$ be a rewriting candidate and let $r^u(\bar{s}, \kappa)$ be the unfolding of r. Let $r_0(\bar{s}, \lambda) \leftarrow B$ be an aggregate query obtained from r by replacing the aggregate term κ with the term λ.*

If $r_0 \equiv_{\mathcal{V}} r^u$, then the functions λ and $f^\kappa(\prod \bar{z})$ compute the same aggregate value for every database and every group of values retrieved by $\breve{r} = \breve{r}_0$.

Proof. (Sketch) The query r is equivalent to r^u modulo $\mathcal{V}$. Hence, $r \equiv_{\mathcal{V}} r_0$. Now the statement follows because aggregates are functionally dependent on the grouping variables. □

We have proposed "natural" candidates for rewriting aggregate queries. We now consider the problem of verifying if a rewriting candidate is in fact a rewriting when given a particular query. We call this problem the *rewriting verification problem.*

Theorem 5.11 (Rewriting Verification Criterion) *Let q be an aggregate query and let r be a rewriting candidate of q over $\mathcal{V}$. Then r is a rewriting of q, i.e. $q \equiv_{\mathcal{V}} r$, if and only if $q \equiv r^u$.*

By the theorem above we have reduced the rewriting verification problem to the problem of equivalence of aggregate queries. This problem was considered in Section 4.

5.4 Disjunctive Rewritings

We relax some of our previous restrictions on the form of a rewriting. We consider rewritings defined as disjunctive queries and rewritings using disjunctive views. We will call such rewritings *disjunctive rewritings.* Such extensions are both natural and necessary to increase the possibilities of finding rewritings.

5.4.1 Definitions

We extend our definition of candidates for disjunctive rewritings using disjunctive views. Essentially, we require that each disjunct be a candidate as defined above (see Definition 5.6). Formally, the definition is as follows.

Definition 5.12 (Disjunctive Rewriting Candidates) *Let α be one of max, sum, or count. A query r is a* disjunctive α-rewriting candidate *if it has the form*

$$r(\bar{s}, \beta(z)) \quad \leftarrow \quad r_1(\bar{s}, z) \vee \ldots \vee r_n(\bar{s}, z),$$

where

1. *$r_1, \ldots, r_n$ are conjunctive α-rewriting candidates;*
2. *$\beta = max$ if α is max, and $\beta = sum$ if α is sum or count.*

Unfolding is defined similarly as in the conjunctive case (Definition 5.8). However we require that the result be transformed to disjunctive normal form (DNF). This is useful since the only known technique for checking equivalence requires the queries to be in DNF. The Rewriting Verification Criterion (Theorem 5.11) still holds. The proof makes use of the result for conjunctive rewritings.

5.4.2 Expressiveness of Disjunctive Rewritings

The general goal of allowing disjunctive rewritings is to increase the expressiveness of rewritings. However, we show that under certain conditions the existence of a disjunctive rewritings implies already the existence of a conjunctive rewriting. By a "pure disjunctive rewriting" we mean a rewriting having only pure candidates as disjuncts.

Theorem 5.13 (Expressiveness of Rewritings) *Let q be a linear count or sum-query over the rational numbers, and let $\mathcal{V}$ be a set of relational conjunctive views. If there exists a pure disjunctive rewriting of q over $\mathcal{V}$, then there exists also a pure conjunctive rewriting of q.*

Proof. (Sketch) Without loss of generality we assume that q is a reduced query. Let r be a rewriting of q over $\mathcal{V}$. Then $\breve{q}$ and $\breve{r^u}$ are bag-set-equivalent. Hence, by Theorem 4.4, they have isomorphic linear expansions over the set of constants appearing in q or r^u.

Let L be a linearization that does not identify terms. Such a linearization exists because q is a query ranging over the rationals. Let $\breve{q}_L$ be the corresponding disjunct in the linear expansion of $\breve{q}$.

There is a query isomorphic to $\breve{q}_L$ in the linear expansion of the core of the unfolding of r. Let this query be $(\breve{r^u_k})_M$, where $\breve{r^u_k}$ is a disjunct of r, and M is a linearization of the core of $\breve{r^u_k}$. Since q_L is linear, there is exactly one isomorphism between the two queries. We assume w.l.o.g. that this isomorphism is the identity.

The core of q has the form $\breve{q}(\bar{x}) \leftarrow R \;\&\; C$, while r_k has the form

$$r_k(\bar{x}, \prod z_i) \leftarrow v_1(\theta_1\bar{x}_1, z_1) \;\&\; \ldots \;\&\; v_l(\theta_l\bar{x}_l, z_l) \;\&\; C_k,$$

and the core of the unfolding of r_k has the form

$$\breve{r^u_k}(\bar{x}) \leftarrow \theta_1 B_1 \;\&\; \ldots \;\&\; \theta_l B_l \;\&\; C_k,$$

where B_j is the core of the view v_j. The views v_j do not have comparisons, therefore each $\theta_j B_j$ is a conjunction of relational atoms. We abbreviate the conjunction of the $\theta_j B_j$ as R_k. Since the identity mapping is an isomorphism between q_L and $(\breve{r^u_k})_M$, the relational parts of the bodies are identical, that is, $R = R_k$.

We call those variables in the body of a query that do not appear in the head of the query *nondistinguished variables*. The variables in R can be partitioned into those that are introduced by a θ_j, and the nondistinguished variables of the bodies of the v_j. One can show that all variables appearing in C, the comparisons of q, are of the first kind. This will complete the proof because then we obtain a conjunctive rewriting of q from r_k by replacing the comparisons C_k in r_k with the comparisons C of q. □

Note that according to our definition of valid views summarized in Table 2, pure disjunctive candidates for rewriting *count* and *sum*-queries use only *count*-views. Note as well, that as the following example demonstrates, the preceding theorem does not hold over the integers.

Example 5.14 Consider the count query q and the set of views $\mathcal{V} := \{v_1, v_2\}$, ranging over the integers and defined as follows:

$$\begin{aligned} q(count) &\leftarrow p(x,y) \;\&\; 1 \le x \le 2 \\ v_1(count) &\leftarrow p(1,y) \\ v_2(count) &\leftarrow p(2,y). \end{aligned}$$

Then $r(sum(z)) \leftarrow v_1(z) \vee v_2(z)$ is a disjunctive rewriting of q, but it is easy to see that there is no conjunctive rewriting of q over $\mathcal{V}$.

Disjunctive rewritings have additional sources of complexity over conjunctive rewritings. Allowing disjunctive queries adds expressiveness and therefore can add complexity. However, we will show (Theorem 7.1) that allowing disjunctive rewritings does not increase the complexity of the rewriting verification problem.

6 Examples

In this section we present examples that illustrate query rewritings. Our examples come from the university environment.

6.1 Example 1

We consider the following university database, containing two schemes and two views on the schemes.

```
grades(student_name,course_name,grade)
courses(course_name,teacher_name,location)

v_max_course_grade(c,max(g))  ← grades(s,c,g)
v_courses_and_teachers(c,t)   ← courses(c,t,l)
```

The `courses` relation describes the courses given in the university. We assume that each course may be taught by a number of teachers and may take place in a number of locations. In addition, each teacher can teach many courses. The `grades` relation contains information on students and their grades in the courses they have taken.

The view `v_max_course_grade` defines a relation which contains the maximal grade in each course, and the view `v_courses_and_teachers` projects the relation `courses` on attributes `course_name` and `teacher_name`.

Suppose we are given the following query:

$$q(c, max(g)) \leftarrow \texttt{courses}(c, \text{“Smith”}, l) \;\&\; \texttt{grades}(s, c, g)$$

This query retrieves the maximal grade in each course taught by Smith. We are interested in rewriting the query with the views given above.

According to Definition 5.2, for rewriting a *max*-query, both non-aggregate and *max*-views are valid. Thus both `v_max_course_grade` and `v_courses_and_teachers` are *valid.*

By Definition 5.5 of *pure candidates*, we can only use the view `v_courses_and_teachers` in searching for a pure candidate. Clearly we can only find a *partial rewriting* of our query using only pure candidates. This rewriting will be of the following form:

$$\begin{aligned} r_1(c', max(g')) \;\leftarrow\; & \texttt{grades}(s, c', g') \;\& \\ & \texttt{v\_courses\_and\_teachers}(c', \text{“Smith”}) \end{aligned}$$

When searching for a *max-view candidate*, we must use the *max*-view, and the y parameter of the candidate's formula (Definition 5.4) must appear only as the return value of it. Thus we can derive two rewritings from *max*-view candidates as follows:

$$\begin{aligned} r_2(c', max(g')) \;\leftarrow\; & \texttt{v\_max\_course\_grade}(c', g') \;\& \\ & \texttt{courses}(c', \text{“Smith”}, l) \\ r_3(c', max(g')) \;\leftarrow\; & \texttt{v\_max\_course\_grade}(c', g') \;\& \\ & \texttt{v\_courses\_and\_teachers}(c', \text{“Smith”}) \end{aligned}$$

Note that r_2 is a partial rewriting, while r_3 is a *complete rewriting*. Also note that all the rewritings presented are *undiminishable*.

By our unfolding technique we can prove that all the rewriting candidates above are in fact rewritings. To simplify we will prove only that r_3 is a rewriting. Reasoning for r_1 and r_2 may be done in an analogous way.

By Theorem 5.11 in order to prove that r_3 is a rewriting of q we have to show that $r_3^u \equiv q$.

We note that the substitution of `v_max_course_grade`, θ_1 is defined as $\{c'/c\}$. Similarly, we define the substitution of `v_courses_and_teachers`, θ_2 as $\{c'/c, \text{“Smith”}/t\}$. Thus the extension of θ_1 is $\hat{\theta}_1 = \{c'/c, g'/g\}$ since g' is the output variable of the instantiation of the view and g is its parameter (Definition 5.7). Clearly, $\hat{\theta}_2 = \theta_2$.

Applying the Definition 5.8 of an unfolding we get

$$\begin{aligned} r_3^u(c', max(g')) \quad \leftarrow \quad & \texttt{grades}(s, c', g')\ \& \\ & \texttt{courses}(c', \text{“Smith”}, l) \end{aligned}$$

In order to complete the proof we have to establish the equivalence between the two *max* queries—q and r_3^u. By the *max* queries equivalence criterion (Theorem 4.7) we must evaluate the cores of the queries and check for mutual dominance. The cores of q and r_3^u are relational and thus we have shown that dominance is exactly containment. The cores of q and r_3^u are isomorphic and therefore they clearly contain one another. Thus $r_3^u \equiv q$ and r_3 is in fact a rewriting.

6.2 Example 2

We now look at another aspect of the university database. We consider the following schemes and views that define relations pertaining to salaries of teaching assistants. This example may seem a bit odd but in fact it models exactly the situation at the Hebrew University in Jerusalem as pertaining to the payment policy of teaching assistants.

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

v_positions_per_type(j,count)  ←  ta(n,c,j)
v_salary_for_ta_job(j,sum(a))  ←  salaries(j,s,a)
```

At the Hebrew University, there may be many teaching assistants in a course and a student may be a TA in many courses. Each TA has a `job_type` in the course he assists. For example, he may give lectures, grade exercises, or instruct a lab. 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. All the data is kept in the two schemes defined above.

In the first view, `v_positions_per_type`, we compute the number of positions of each type held in the university. In the other view, `v_salary_for_ta_job` we compute the total salary given for each type of position.

We are interested in calculating the total amount of money spent on each job position. This can be evaluated by the following query:

$$q(j, sum(a)) \leftarrow \texttt{ta}(n, c, j)\ \&\ \texttt{salaries}(j, s, a)$$

Both views defined above are valid for rewriting this query. It may be noted that these views are not valid for rewriting the query in the previous example, and vice versa.

To create a pure candidate we can only use the view `v_positions_per_type` and clearly we can only create a partial rewriting using this view, as it does not contain any information from the `salaries` relation.

We will present the only undiminishable complete rewriting that may be computed using the above defined views. It is derived as a *sum* view candidate.

$$\begin{aligned} r(j', sum(a' * cnt)) \quad \leftarrow \quad & \texttt{v\_positions\_per\_type}(j', cnt)\ \& \\ & \texttt{v\_salary\_for\_ta\_job}(j', a') \end{aligned}$$

To prove that this candidate is in fact a rewriting we must unfold r. We perform an unfolding deriving the following query:

$$r^u(j', sum(a')) \leftarrow \texttt{ta}(n, c, j')\ \&\ \texttt{salaries}(j', s, a')$$

Clearly, r^u and q are isomorphic and thus it follows that $r^u \equiv q$. We have shown that the unfolding operation preserves equivalence and thus r is a rewriting.

7 Complexity Results

7.1 Rewriting Verification Problem

Given a query q, and a rewriting candidate, r, recall that the rewriting verification problem is the problem of checking whether r is in fact a rewriting of q. We have shown that we can reduce rewriting verification to equivalence of aggregate queries (Theorem 5.11). This reduction is done by unfolding the candidate, to derive an aggregate query r^u and checking equivalence of q and r^u.

Conjunctive queries can be unfolded in polynomial time. Thus in this case, the rewriting verification problem has exactly the same complexity as checking for equivalence. Therefore, the complexity results obtained in Table 1 for equivalence can be easily viewed as complexity results for the rewriting verification problem when we replace q' with r^u.

Verifying disjunctive rewritings has an additional source of complexity over conjunctive rewritings. Allowing disjunctive queries adds expressiveness and one may therefore suppose that it adds complexity. An additional source of complexity is the natural extension of the unfolding technique for disjunctive rewriting candidates. Recall that one step in computing the unfolding and checking for equivalence is to create a query in DNF. Thus unfolding a candidate may yield an exponential blowup in the size of the query if the candidate contains disjunctive views and one may conjecture that verification of disjunctive rewritings and searching for a rewriting may be much more difficult in this case. However, we have shown that our upper bounds for the problem of verifying conjunctive rewritings also hold for the disjunctive case.

Theorem 7.1 (Complexity of Verification)

- *One can verify in* PSPACE *whether a disjunctive candidate is a rewriting of a* count *or a* sum*-query.*
- *For* max*-queries the problem to verify whether a disjunctive candidate is a rewriting is* Π_2^{P}*-complete.*

The intuitive reason why the above theorem holds is that some kind of disjunction is already present in the verification problem for queries with comparisons. The additional source of complexity due to disjunctive views is of the same kind as the one introduced by comparisons.

7.2 Rewriting Existence Problem

Given a query q and a set of views $\mathcal{V}$ we consider the problem of checking whether there exists a complete or partial rewriting of q using $\mathcal{V}$, in the form defined above. We call these problems the *complete* or *partial rewriting existence problem* respectively.

Compared to the Verification Problem, the Existence Problem has an additional source of complexity, since here the candidate to be verified also has to be found.

For relational *max*-queries and relational views, verification essentially consists in finding containment mappings between the query and the unfolded candidate (cf. Theorem 4.8). Therefore, the need to also find the rewriting candidate does not add to the overall complexity of the problem.

Theorem 7.2 (Relational Max-Queries)

1. *Both the complete and partial rewriting existence problems are* NP*-complete for relational* max*-queries and sets of relational conjunctive views.*
2. *Both problems are still* NP*-complete even if both, queries and views, are linear and conjunctive.*

For *sum* and *count*-queries we consider first the case where the query, q, is conjunctive and where we are interested in checking the existence of a conjunctive rewriting, r, over a set of conjunctive views $\mathcal{V}$. Our results are summarized in Table 4 below.

query type	complete rewriting	partial rewriting
arbitrary	NP-hard/in PSPACE	in PSPACE
q is relational	NP-complete	NP-complete
q is linear	NP-complete	polyn.

Table 4: Complexity of the Rewriting Existence Problem for *sum* and *count*-queries in the conjunctive case.

The difficulty in proving decidability in the general case is that a query may have rewritings of arbitrary size, as is illustrated by the following example.

Example 7.3 Consider the *count*-query q and the view v defined as follows:

$$\begin{array}{rcl} q(x, count) & \leftarrow & p(x,y) \ \& \ 0 \leq x \\ v(x, count) & \leftarrow & p(x,y) \ \& \ 0 \leq x. \end{array}$$

Then, obviously, $r(x,z) \leftarrow v(x,z)$ is a rewriting of q. However, if we define queries r_i^n for $i \in 1..n$ as

$$\begin{array}{rcl} r_i^n(x, count) & \leftarrow & p(x,y) \ \& \ i \leq x \ \& \ x \leq i+1 \\ & & \qquad \text{if } 0 \leq i < n \\ r_n^n(x, count) & \leftarrow & p(x,y) \ \& \ n \leq x, \end{array}$$

then also

$$r^n(x, sum(z)) \leftarrow r_0^n(x,z) \vee \ldots \vee r_n^n(x,z)$$

is a rewriting of q for every n. The r^n may be arbitrarily large. However, in order to form them, new constants are needed that occur neither in the query nor the views, namely the numbers, $1, 2, \ldots, n$.

We conjecture that if there is a rewriting of a query q using $\mathcal{V}$, then there exists one that uses only constants occurring in q or in $\mathcal{V}$. However, we were not able to prove this. We were only able to show decidability for the case that we consider only rewritings that do *not* introduce new constants.

Theorem 7.4 (Decidability of Restricted Existence Problem) *Let q be an aggregate query and $\mathcal{V}$ be a set of views. Under the assumption that we are only searching for a rewriting using constants appearing in q or $\mathcal{V}$, the rewriting existence problems are decidable.*

8 Conclusion

Questions related to the view usability problem have been studied extensively. For example, containment and equivalence under set semantics, have been investigated in [CM77, ASU79, SY81, JK83, SS92, vdM92, LMSS93, LS95]. Containment for conjunctive queries under multiset semantics, which is the semantics of SQL, has been studied in [CV93].

Techniques for using views to answer queries have been suggested by a number of researchers, although most of this work did not pay much attention to the formal aspects of the problem [YL87, CR94, CKPS95].

The view usability problem for conjunctive queries under set semantics has been treated by Levy et al. in [LMSS95]. Chaudhuri et al. investigated view usability for conjunctive queries under multiset semantics in [CKPS95].

A method to use views for queries with grouping and aggregates has been developed by Gupta et al. [GHQ95]. The method is based on rewrite rules to transform the tree representation of a query. It is sound, but it does not allow one to find all possible equivalent rewritings using the views. Levy et al. studied the same problem and gave sufficient conditions for an aggregate SQL-query to be computable from a set of views. Their algorithms are claimed to be complete in some cases, e.g., when the views do not contain aggregation and the constraints in the where-part of the query and the views contain only equality predicates, although no proofs are provided [SDJL96].

We have studied the problem of finding rewritings of aggregate queries using aggregate and non-aggregate views. Our approach is based on syntactic characterizations of the equivalence of aggregate queries.

We defined classes of aggregate queries using views for which unfolding the view definitions is a transformation that preserves equivalence. Such queries are natural candidates for rewritings. A candidate is a rewriting of a given query if the query and the unfolding of the candidate are equivalent. Thus syntactic characterizations of equivalence turn into syntactic characterizations of rewritings.

The characterizations are the basis for studying the problem of *finding* rewritings. This problem has two sources of complexity, the first of which is to assemble the rewriting, and the second to verify that it is in fact a rewriting.

We distinguish between complete and partial rewritings, the former being to rewrite the query using *only* views. It turns out that in general the two problems have the same complexity, although for special cases, finding partial rewriting is an easier problem.

We leave for future research the problem of rewriting w.r.t. integrity constraints, the rewriting of nested queries, or queries with a `having` clause. Rewriting these queries involves open questions in equivalence theory.

9 Acknowledgments

This research was supported in part by the Esprit Long Term Research Project 22469 "Foundations of Data Warehouse Quality" (DWQ), by Grants 8528-95-1 and 9481-1-98 of the Israeli Ministry of Science, and by a grant of the French Ministry of Research and Technology.

The authors express their gratitude to Yehoshua Sagiv for many valuable discussions.

References

[ASU79] A.V. Aho, Y. Sagiv, and J.D. Ullman. Efficient optimization of a class of relational expressions. *ACM Transactions on Database Systems*, 4(4):435–454, 1979.

[CKPS95] S. Chaudhuri, S. Krishnamurthy, S. Potarnianos, and K. Shim. Optimizing queries with materialized views. In P.S.Yu and A.L.P. Chen, editors, *Proc. 11th International Conference on Data Engineering*, Taipei, March 1995. IEEE Computer Society.

[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.

[CR94] C.M. Chen and N. Roussopoulos. The implementation and performance evaluation of the ADMS query optimizer. In M. Jarke, editor, *Proc. 4th International Conference on Extending Database Technology*, Cambridge (UK), March 1994. Springer-Verlag.

[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.

[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.

[IR95] Y.E. Ioannidis and R. Ramakrishnan. Beyond relations as sets. *ACM Transactions on Database Systems*, 20(3):288–324, 1995.

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

[Kim96] R. Kimball. *The Data Warehouse Toolkit*. John Wiley and Sons, 1996.

[LFS97] F. Llirbat, F. Fabret, and E. Simon. Eliminating costly redundant computations from SQL trigger executions. In *Proc. 1997 ACM SIGMOD International Conference on Management of Data*, pages 428–439, Tucson (Arizona, USA), June 1997.

[LMSS93] A.Y. Levy, I. Singh Mumick, Y. Sagiv, and O. Shmueli. Equivalence, query-reachability, and satisfiability in datalog extensions. In *Proc. 12th Symposium on Principles of Database Systems*, pages 109–122, Washington (D.C., USA), May 1993. ACM Press.

[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.

[LS95] A.Y. Levy and Y. Sagiv. Semantic query optimization in datalog programs. In *Proc. 14th Symposium on Principles of Database Systems*, pages 163–173, San Jose (California, USA), Proc. 14th Symposium on Principles of Database Systems 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 *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.

[RSS96] K.A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *Proc. 1996 ACM SIGMOD International Conference on Management of Data*, pages 447–458, Montreal (Canada), June 1996.

[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.

[SS92] Y. Sagiv and Y. Saraiya. Minimizing restricted-fanout queries. *Discrete Applied Mathematics*, 40:245–264, 1992.

[SY81] Y. Sagiv and M. Yannakakis. Equivalence among relational expressions with the union and difference operators. *J. ACM*, 27(4):633–655, 1981.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume I. Computer Science Press, 1988.

[vdM92] R. van der Meyden. The complexity of querying indefinite data about linearly ordered domains. In *Proc. 11th Symposium on Principles of Database Systems*, pages 331–345, San Diego (California, USA), May 1992. ACM Press.

[YL87] H.Z. Yang and P.-A. Larson. Query transformation for PSJ queries. In *Proc. 13th International Conference on Very Large Data Bases*, pages 245–254, Brighton (England), September 1987. Morgan Kaufmann Publishers.