Algorithms for Rewriting Aggregate Queries Using Views

Sara Cohen
Werner Nutt
Alexander Serebrenik

Report CW 292, May 2000

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Algorithms for Rewriting Aggregate Queries Using Views

Sara Cohen
Werner Nutt
Alexander Serebrenik

Report CW 292, May 2000

Department of Computer Science, K.U.Leuven

Abstract

Queries involving aggregation are typical in database applications. 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. Due to a lack of theory, algorithms for this problem were rather ad-hoc. They were sound, but were not proven to be complete.

Recently we have given syntactic characterizations for the equivalence of aggregate queries and applied them to decide when there exist rewritings. However, these decision procedures do not lend themselves immediately to an implementation. In this paper, we present practical algorithms for rewriting queries with *count* and *sum*. Our algorithms are sound. They are also complete for important cases. Our techniques can be used to improve well-known procedures for rewriting non-aggregate queries. These procedures can then be adapted to obtain algorithms for rewriting queries with *min* and *max*. The algorithms presented are a basis for realizing optimizers that rewrite queries using views.

Algorithms for Rewriting Aggregate Queries Using Views

Sara Cohen
Computer Science Department
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
Computer Science Department
K. U. Leuven
Celestijnenlaan 200A,
B-3001 Heverlee, Belgium
Alexander.Serebrenik@cs.kuleuven.ac.be

Technical Report CW 292

Algorithms for Rewriting Aggregate Queries Using Views

Abstract. Queries involving aggregation are typical in database applications. 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. Due to a lack of theory, algorithms for this problem were rather ad-hoc. They were sound, but were not proven to be complete.
Recently we have given syntactic characterizations for the equivalence of aggregate queries and applied them to decide when there exist rewritings. However, these decision procedures do not lend themselves immediately to an implementation. In this paper, we present practical algorithms for rewriting queries with *count* and *sum*. Our algorithms are sound. They are also complete for important cases. Our techniques can be used to improve well-known procedures for rewriting non-aggregate queries. These procedures can then be adapted to obtain algorithms for rewriting queries with *min* and *max*. The algorithms presented are a basis for realizing optimizers that rewrite queries using views.

1 Introduction

Aggregate queries occur in many applications, such as data warehousing [TS97], mobile computing [BI94], and global information systems [LRO96b]. The size of the database in these applications is generally very large. Aggregation is often used in queries against such sources as a means of reducing the granularity of data. The execution of aggregate queries tends to be time consuming and costly. Computing one aggregate value often requires scanning 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. If the previous queries involved aggregation, the answers to them will tend to be much smaller than the size of the database. Thus, using their answers will be much more efficient.

We call a reformulation of a query that uses other queries a *rewriting.* Finding such rewritings is known 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.* 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 the rewriting.

Rewriting queries using views has been studied 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 [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, syntactic characterizations for the equivalence of SQL queries with the aggregate operators *min*, *max*, *count*, and *sum* have been given [NSS98]. They have been applied 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].

Using these characterizations, one can "guess" candidates for rewritings and verify if they are in fact equivalent to the original query. However, this process is highly nondeterministic. Clearly, it is more efficient to gradually build a candidate for rewriting in a way that will ensure its being a rewriting. The characterizations do not immediately yield practical algorithms of this sort. In fact, there are several subtle problems that must be dealt with in order to yield complete algorithms.

In this paper, we show how to derive practical algorithms for rewriting aggregate queries. 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. In Appendix A we demonstrate how queries written in SQL can be translated to our extended Datalog syntax and vice versa.

2 Motivation

We discuss an example that illustrates the rewriting problem for aggregate queries. All the examples in this paper are written using an extended Datalog syntax. This syntax is more abstract and concise than SQL. In Section 3 we present a formal definition of the Datalog syntax. In Appendix A we describe how queries written in SQL can be translated to our Datalog syntax and vice versa.

The following 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)` and
`salaries(job_type,sponsorship,amount)`.

At the Hebrew University, there may be many teaching assistants in a course. Each TA has at least one `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 one of them, `v_positions_per_type`, we compute the number of positions of each type held in the university. In the second view, `v_salary_for_ta_job` we compute the total salary for each type of position. We express aggregate queries with an extended Datalog notation, where in the head we separate grouping variables and aggregate terms by a semicolon:

$$\texttt{v\_positions\_per\_type}(j; \mathit{count}) \leftarrow \texttt{ta}(n, c, j)$$
$$\texttt{v\_salary\_for\_ta\_job}(j; \mathit{sum}(a)) \leftarrow \texttt{salaries}(j, s, a).$$

In Subsection 3.2 we define a semantics for such Datalog queries that identifies them with SQL queries where the attributes in the `GROUP BY` clause and those in the `SELECT` clause coincide. The grouping variables correspond to those attributes.

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

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

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:

$$r(j'; a' * \mathit{cnt}) \leftarrow \texttt{v\_positions\_per\_type}(j'; \mathit{cnt})\ \&\ \texttt{v\_salary\_for\_ta\_job}(j'; a')$$

In order to evaluate the new query, we no longer 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. Thus, we will only be able to build an optimizer that can rewrite aggregate queries, if we can provide answers to the following two questions.

- **Rewriting Verification:** How can we prove that a new query, which uses views, produces the same results as the original query?
- **Rewriting Computation:** 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. We extend the well-known Datalog syntax for non-aggregate queries [Ull89] so that it covers also aggregates. These queries express nonnested SQL 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 and 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 either $<$ or $\leq$. If C and C' are conjunctions of comparisons, we write $C \models C'$ if C' is a consequence of C. 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 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 every predicate symbol p a relation $p^{\mathcal{D}}$, that is, a set of tuples. Under *set semantics*, a conjunctive query q defines a new relation $q^{\mathcal{D}}$, which consists of all the answers that q produces over $\mathcal{D}$. Under *bag-set semantics*, q defines a multiset or *bag* $\{\!\{q\}\!\}^{\mathcal{D}}$ of tuples. 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 [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.

3.2 Aggregate Queries

We now extend the Datalog syntax so as to capture also queries with `GROUP BY` and aggregation. We assume that queries have only one aggregate term. The general case can easily be reduced to this one [CNS99]. 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.[1] For example, *count* and $sum(z_1 * z_2)$, 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. Note that $\bar{y}$ is empty if κ is the *count* aggregation function. Terms of the form *count*, $sum(y)$ and $max(y)$ are *elementary aggregate terms*. Abstractly, elementary aggregate terms are denoted as $\alpha(y)$, where α is an aggregation 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, $sum(z_1 * z_2)$ describes the aggregation function $f_{sum(z_1*z_2)}$ that maps any multiset M of pairs of numbers (m_1, m_2) to $\sum_{(m_1,m_2)\in M} m_1 * m_2$.

An *aggregate query* is a conjunctive query augmented by an aggregate term in its head. Thus, it has the form $q(\bar{x}; \kappa(\bar{y})) \leftarrow B(\bar{s})$. 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

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

That is, the bag $\Gamma_{\bar{d}}$ 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, $\Gamma_{\bar{d}}$ is the multiset of $\bar{y}$-values that $\breve{q}$ returns for $\bar{d}$. The result of evaluating q over $\mathcal{D}$ is

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

[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.

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 until now, we have defined equivalence of aggregate queries and equivalence of non-aggregate queries under set and bag-set-semantics. 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 database relations, but are determined by the base relations indirectly. In order to take this relationship into account, we define 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. 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

We give a general definition of rewritings. 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.

- **Rewriting Verification:** Given queries q and r, and a set of views $\mathcal{V}$, check whether $q \equiv_{\mathcal{V}} r$.
- **Rewriting Computation:** 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 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 Reminder: 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 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.[2] Thus, a candidate for rewriting $q(\bar{x})$ 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 occurrences of the 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 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,*

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

The *third question* is how to check whether r is a rewriting of q, that is, whether r and q are *equivalent modulo* $\mathcal{V}$. This can be achieved by checking whether r^u and q are set-equivalent: if $r^u \equiv q$, then the Unfolding Theorem implies $r \equiv_{\mathcal{V}} q$. Set-equivalence of conjunctive queries can be decided syntactically by checking whether there are homomorphisms in both directions [Ull89].

4.2 Rewritings of 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 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

[2] 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.

do not lose multiplicities.[3] 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 1. Consider the query

$$\texttt{q\_positions\_per\_type}(j; count) \leftarrow \texttt{ta}(n, c, j)$$

in which we compute the number of positions of each type held in the university. Recall the view `v_positions_per_type` defined in Section 2. It is easy to see that both of the following are rewritings of `q_positions_per_type`:

$$r_1(j'; z) \leftarrow \texttt{v\_positions\_per\_type}(j'; z)$$
$$r_2(j'; \sqrt{z_1 * z_2}) \leftarrow \texttt{v\_positions\_per\_type}(j'; z_1)\ \&\ \texttt{v\_positions\_per\_type}(j'; z_2).$$

By adding additional view atoms and adjusting the power of the root we can create infinitely many different rewritings of `q_positions_per_type`. It is natural to create only r_1 as a rewriting of q. In fact, only for r_1 will the Unfolding Theorem hold.

We define a *candidate* for a rewriting of $q(\bar{x}; count) \leftarrow R\ \&\ C$ as a query having the form

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

where v_i^c are *count*-views, possibly with comparisons, defined as $v_i^c(\bar{x}_i; count) \leftarrow B_i$ and z_i are variables not appearing elsewhere in the body of r. We call r a *count*-rewriting candidate.

Note that it is possible to omit the summation if the values of z_i are functionally dependent on the value of the grouping variables $\bar{x}$. This is the case, if only grouping variables appear as $\theta_i x_i$ in the heads of the instantiated views. Then the summation is always over a singleton group.

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. In the unfolding, we replace the view atoms of the rewriting with the appropriate instantiations of their bodies, and we replace the aggregate term in the rewriting with the term *count*. Thus, we obtain as the unfolding r^u of r the query

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

[3] 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.

In [CNS99], it has been proven that for r^u the Unfolding Theorem holds, i.e., $r \equiv_{\mathcal{V}} r^u$. Moreover, it has been shown that this definition of unfolding uniquely determines the aggregation function in the head of our candidates. That is, summation over products of counts is the only aggregation function for which the Unfolding Theorem holds if r^u is defined as above. Now, in order to verify that r is a rewriting of q, we can check that r^u is equivalent to r, without taking into account the views any more.

We now present an algorithm that finds a rewriting for a *count*-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.

In [NSS98], a sound and complete characterization of equivalence of conjunctive *count*-queries with comparisons has been given. 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, it has been shown [CV93,NSS98] that two relational *count*-queries are equivalent if and only if they are isomorphic. In addition, 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 relational *count*-queries and of linear *count*-queries. This algorithm is sound and tractable for the general case, but is not complete.

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 $q(\bar{x}; count) \leftarrow R \,\&\, C$, that is, when the instantiated view can occur in a partial rewriting. By the characterization of equivalence for relational and linear queries, 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 θR_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 $\theta\bar{u}$ (i.e., the nondistinguished variables in v) 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. We capture this property by defining that $v(\bar{u}; count) \leftarrow R_v \,\&\, C_v$ is *R-usable under θ w.r.t. φ* if

1. $\varphi\theta R_v$ is isomorphic to a subset R' of R, and
2. every variable that occurs both in R' and in $R \setminus R'$ must occur in $\theta\bar{u}$.

We denote this fact as R-usable(v, θ, φ). Clearly, there is a partial rewriting using v of $q(\bar{x}; count) \leftarrow R \,\&\, C$ only if $v(\bar{u}; count) \leftarrow R_v \,\&\, C_v$ is R-usable under θ w.r.t. some φ.

Example 2. Consider the following query that computes the number of sponsors for each assistant in the database course

$$\texttt{q\_db\_ta\_sponsors}(n; count) \leftarrow \texttt{ta}(n, \texttt{Database}, j) \,\&\, \texttt{salaries}(j, s, a).$$

We suppose that we have a materialized view that computes the number of jobs that each teaching assistant has in each course that he assists

$$\texttt{v\_jobs\_per\_ta}(n', c'; count) \leftarrow \texttt{ta}(n', c', j').$$

In order to use `v_jobs_per_ta` in rewriting `q_db_ta_sponsors` we must find an instantiation θ such that $\theta\texttt{ta}(n', c', j')$ covers some part of the body of `q_db_ta_sponsors`. Clearly, $\theta\texttt{ta}(n', c', j')$ can cover only $\texttt{ta}(n, \texttt{Database}, j)$. We take, $\theta = \{n'/n, c'/\texttt{Database}\}$ and thus, $\varphi = \{n/n, j'/j\}$. However, j appears in $\texttt{ta}(n, \texttt{Database}, j)$ and not in the head of $\theta\texttt{v\_jobs\_per\_ta}$ and therefore, j is not accessible after replacement. Note that j appears in `salaries` and thus, `v_jobs_per_ta` is not R-usable in rewriting `q_db_ta_sponsors`.

For our algorithm to be complete for linear queries, the set of comparisons in the query to be rewritten has to be deductively closed (see Example 4). The deductive closure of a set of comparisons can be computed in polynomial time [Klu88]. In addition, it must hold that $C \models \varphi(\theta C_v)$, thus, the comparisons inherited from v are weaker than those in q. For a rewriting using θv to exist it must be possible to strengthen $\varphi(\theta C_v)$ by additional comparisons C' so that $\varphi(\theta C_v)$ & C' is equivalent to C. 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 $\varphi(\theta C_v)$ to imply all the comparisons in q that contain an image of a nondistinguished variable in v. Formally, let $ndv(v)$ be the set of nondistinguished variables in v. Let $C^{\varphi(\theta ndv(v))}$ consist of those comparisons in C that contain variables in $\varphi(\theta ndv(v))$. Then, in order for θv to be usable in a partial rewriting, $C_v \models C^{\varphi(\theta ndv(v))}$ must hold. If this condition and $C \models \varphi(\theta C_v)$ hold, then we say that v *is C-usable under θ w.r.t. φ* and write C-usable(v, θ, φ).

We summarize the discussion in a theorem.

Theorem 2. *Let $q(\bar{x}; count) \leftarrow R$ & C be a count-query whose set of comparisons C is deductively closed, and let $v(\bar{u}; count) \leftarrow R_v$ & C_v be a count-view. There exists a partial rewriting of q using v if and only if there is a φ such that R-usable(v, θ, φ) and C-usable(v, θ, φ).*

Example 3. The following query computes for each job the number of mediocre sponsors, i.e., the number of sponsors who give an amount that is greater than \$200 and less than \$600.

$$\texttt{q\_mediocre\_sponsor}(j; count) \leftarrow \texttt{salaries}(j, s, a)\ \&\ a > 200\ \&\ a < 600.$$

The view

$$\texttt{v\_all\_sponsor}(j'; count) \leftarrow \texttt{salaries}(j', s', a')\ \&\ a' > 0$$

computes the number of sponsors for each job. In order to use `v_all_sponsor` in rewriting `q_mediocre_sponsor` we clearly must take $\theta = \{j'/j\}$ and $\varphi = \{j/j, s'/s, a'/a\}$. It holds that $\{a > 200\ \&\ a < 600\} \models \{\varphi\theta(a' > 0)\}$. Observe that a' is a nondistinguished variable in `v_all_sponsor` and a' is mapped

to a by $\varphi\theta$. Thus, in order for `v_all_sponsor` to be C-usable for rewriting `q_mediocre_sponsor` it must hold that $\{\varphi\theta(a' > 0)\} \models \{a > 200 \,\&\, a < 600\}$. This does not hold. Therefore, `v_all_sponsor` is not C-usable for rewriting `q_mediocre_sponsor`.

We present an algorithm for computing rewritings of conjunctive *count*-queries in Figure 1. The underlying idea is to incrementally cover the body of the query by views until no atom is left to be covered. The algorithm nondeterministically chooses a view v and an instantiation θ, such that v is both R-usable and C-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. Of course, the nondeterminism in choosing the views can be further reduced, for instance, by imposing an ordering on the atoms in the body of the query and by trying to cover the atoms according to that ordering. Note, that the same algorithm may be used to produce partial rewritings if we relax the termination condition of the while-loop. This will similarly hold for subsequent algorithms presented.

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.

Observe 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. These comparisons are not lost because v is C-usable. The comparisons remaining in C are needed to strengthen those inherited from the views such that they are equivalent to the comparisons in the query to be rewritten.

Count_Rewriting is both sound and complete for linear queries and relational queries and is sound, but not complete, for arbitrary queries. Our algorithm runs in nondeterministic polynomial time by guessing views and instantiations and verifying in polynomial time that the obtained result is a rewriting. For relational queries this is optimal, since checking whether there exists a θ such that v is R-usable under θ is NP-hard, which can be shown by a reduction of the graph matching problem. Since for linear queries q and views v the existence of θ and φ such that R-usable(v, θ, φ) and C-usable(v, θ, φ) can be decided in polynomial time, one can obtain a polynomial time variant of the algorithm that computes partial rewritings in the linear case.

Theorem 3. (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 $r \equiv_{\mathcal{V}} q$. If q is either linear or relational, then the opposite holds by making the appropriate choices.*

Example 4. This example shows the incompleteness of the algorithm if the comparisons in the query being rewritten are not deductively closed. Consider the

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, θ, and an isomorphism φ, such that R-usable(v, θ, φ) and C-usable(v, θ, φ).
(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** $r(\bar{x}; sum(\prod_{i=1}^{n} z_i)) \leftarrow Rewriting \,\&\, C$.

Fig. 1. Count Query Rewriting Algorithm

following query q, and views v_1 and v_2, defined as

$$\begin{aligned} q(count) &\leftarrow p_1(x) \,\&\, p_2(y) \,\&\, x < y \,\&\, y < 2 \,\&\, \\ &\qquad p_3(u) \,\&\, p_4(w) \,\&\, u < w \,\&\, w < 2 \\ v_1(x, u; count) &\leftarrow p_1(x) \,\&\, p_2(y) \,\&\, x < y \,\&\, y < 2 \,\&\, p_3(u) \,\&\, u < 2 \\ v_2(x, u; count) &\leftarrow p_3(u) \,\&\, p_4(w) \,\&\, u < w \,\&\, w < 2 \,\&\, p_1(x) \,\&\, x < 2. \end{aligned}$$

Note that the comparisons in q are not deductively closed since q does not contain $x < 2$ and $u < 2$. The algorithm Count_Rewriting will not find any rewritings of q using v_1 and v_2. We can understand this in the following way. Suppose that the view v_1 is chosen first. Clearly, v_1 can be used for a rewriting taking the instantiation θ and the isomorphism φ to be the identity mappings. The algorithm Count_Rewriting removes from q the comparisons $x < y$ and $y < 2$ since they contain the variable y which is an image of the nondistinguished variable

y in v_1. However, Count_Rewriting can no longer use v_2 in the rewriting since the constraints in q no longer imply the constraint $x < 2$ in v_2. For symmetric reasons, Count_Rewriting would also fail to find a rewriting if v_2 was chosen first. However, clearly the following is a rewriting of q using v_1 and v_2:

$$r(sum(z_1 * z_2)) \leftarrow v_1(x, u; z_1)\ \&\ v_2(x, u; z_2).$$

Example 5. The algorithm is incomplete for the general case. Consider the following query q, and view v

$$q(; count) \leftarrow p(x)\ \&\ p(y)\ \&\ p(u)\ \&\ x < y\ \&\ x < u$$
$$v(; count) \leftarrow p(x')\ \&\ p(y')\ \&\ p(u')\ \&\ x' < y'\ \&\ u' < y'$$

Clearly q and v are equivalent [NSS98]. However, for all homomorphisms φ from v to q, it holds that $\{x < y\ \&\ x < u\} \not\models \{\varphi(x' < y')\ \&\ \varphi(u' < y')\}$. Thus, v is not C-usable for rewriting q and 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 summation variable into consideration. We present an algorithm for rewriting *sum*-queries that is based on 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 for *sum*-queries:

$$r_1(\bar{x}; sum(y * \prod_{i=1}^{n} z_i)) \leftarrow v_1^c(\theta_1\bar{x}_1; z_1)\ \&\ \ldots\ \&\ v_n^c(\theta_n\bar{x}_n; z_n)\ \&\ C' \quad (1)$$

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

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 1 must appear among $\theta_i\bar{x}_i$ for some i. In [CNS99], it has been shown that if a rewriting candidate is equivalent modulo the views 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:

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

Now, instead of checking whether r is a rewriting of q we can verify whether r^u is equivalent to r. The only known algorithm for checking equivalence of *sum*-queries, presented in [NSS98], requires an exponential blowup of the 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 1 for *sum*-queries.

We first extend the algorithm in Figure 1, 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 2. The algorithm runs in nondeterministic polynomial time.

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

Fig. 2. Sum Query Rewriting Algorithm

Theorem 4. (Soundness and Completeness of Sum Rewriting) *Let q be a sum-query and $\mathcal{V}$ be a set of views. If r is returned by* Sum_Rewriting$(q, \mathcal{V})$, *then r is a sum-rewriting candidate of q and $r \equiv_{\mathcal{V}} q$. If q is 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, we 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:

$$r_1(\bar{x}; max(y)) \leftarrow v_1(\theta_1\bar{x}_1)\ \&\ \ldots\ \&\ v_n(\theta_n\bar{x}_n)\ \&\ C' \tag{3}$$

$$r_2(\bar{x}; max(y)) \leftarrow v^m(\theta_m\bar{x}_m; y)\ \&\ v_1(\theta_1\bar{x}_1)\ \&\ \ldots\ \&\ v_n(\theta_n\bar{x}_n)\ \&\ C' \tag{4}$$

Note that the v_i's are nonaggregate views and that v^m is a *max*-view. 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] it has been shown that if a rewriting candidate is equivalent to its unfolding then it must have one of the above forms.

Reasoning about rewriting candidates can be reduced to reasoning about *max*-queries, by extending the unfolding technique. It has been shown [NSS98] that equivalence of relational *max*-queries is equivalence of their cores. There is a similar reduction for the general case. Algorithms for checking set-equivalence of queries can easily be converted to algorithms for checking equivalence of *max*-queries. Thus, 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]. One well-known algorithm for computing rewritings of queries is the *buckets algorithm* [LRO96b,LRO96a]. Consider a query $q(\bar{x}) \leftarrow R\ \&\ C$. The algorithm creates a "bucket" for each atom $p(\bar{z})$ in R. Intuitively, this bucket contains all the views whose bodies can cover $p(\bar{z})$. The algorithm places into this bucket all the views $v(\bar{y}) \leftarrow R_v\ \&\ C_v$ such that R_v contains an atom $p(\bar{w})$ that can be mapped by some mapping φ to $p(\bar{z})$ such that $C\ \&\ \varphi C'$ is consistent. Next, all combinations of taking a view from each bucket are considered in the attempt to form a rewriting.

Note that by reasoning similarly as in the case of *count* and *sum*-queries, we can reduce the number of views put into each bucket, thus improving on the performance of the algorithm. Suppose there is a nondistinguished variable $w \in \bar{w}$ mapped to $z \in \bar{z}$ and there is an atom containing z in q that is not covered by φR_v. In such a case, if v is used in a rewriting candidate there will not exist a homomorphism from the unfolded rewriting to q such that the body of v covers $p(\bar{z})$. However, a rewriting candidate r is equivalent to a query q if and only if there exist homomorphisms from r^u to q and from q to r^u. Thus, v should not be put in the bucket of $p(\bar{z})$.

Observe that this condition is a relaxed version of the R-usability requirement that ensures the existence of an isomorphism. Clearly this restriction filters out the possible rewritings of q, thereby improving the performance of the buckets algorithm. Thus, our methods for finding rewritings of aggregate queries may be relaxed and used to improve the performance of algorithms for rewriting relational queries. These, in turn, may be modified to rewrite *max*-queries.

5 Conclusion

Aggregate queries are increasingly prevalent due to the widespread use of data warehousing and related applications. They are generally computationally expensive since they scan many data items, while returning few results. 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 using views has been studied for the special case of datacubes [HRU96,Dyr96]. However, there was little theory for general aggregate queries. 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 have been implemented in SICStus Prolog.

References

[BI94] D. Barbara and T. Imielinski. Sleepers and workaholics: Caching strategies in mobile environments. In *Proc. 1994 ACM SIGMOD International Conference on Management of Data*, Minneapolis (Minnesota, USA), May 1994. ACM Press.

[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.

[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.

[LRO96a] Alon Levy, Anand Rajamaran, and Joann Ordille. Querying heterogeneous information sources using source description. In *Proceedings of the 22nd VLDB Conference Mumbai(Bombay), India*, 1996.

[LRO96b] A.Y. Levy, A. Rajaraman, and J.J. Ordille. Query answering algorithms for information agents. In *Thirteenth National Conf. on Artificial Intelligence, AAAI-96*, 1996.

[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.

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

A Translating SQL to Datalog

In this paper we extended 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.

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*. In this section we demonstrate, using examples, how an SQL query of this type can be transformed into one in our extended Datalog notation.

We first show how to transform an SQL query without aggregation into one in Datalog notation. Consider a query with `SELECT`, `FROM`, and `WHERE` clauses. For each relation name in the `FROM` clause we introduce a predicate name, 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.

We demonstrate the translation using an example. This example can easily be generalized to arbitrary SQL queries without `GROUP-BY` and `HAVING` clauses. 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_govt`. For the relation names `ta` and `salaries` we introduce the predicate names `ta` and `salaries`. In the fashion described above, we derive the following equivalent Datalog query:

$$\texttt{q\_govt}(n) \leftarrow \texttt{salaries}(j, \texttt{Govt.}, a)\ \&\ \texttt{ta}(n, c, j)\ \&\ a > 500.$$

In this paper we extended the Datalog syntax so as to capture also queries with `GROUP BY` and aggregation. Using our notation, we can represent 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.

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.

To demonstrate this translation, recall the query in Section 2 that calculates the total amount of money spent on each job type. The following SQL query can be transformed into the previously mentioned Datalog query:

```
SELECT  ta.job_type, sum(amount)
FROM    ta, salaries
WHERE   ta.job_type = salaries.job_type.
```

We have demonstrated how to translate SQL into Datalog. Obviously, the translation from Datalog to SQL can be performed in a similar fashion. Roughly speaking, we replace predicates with relations and variables with attributes. The variables in the head of the Datalog query become attributes in the SELECT clause of the SQL query and the comparisons are placed in the WHERE clause. Hence, one notation can be transformed into the other, back and forth, completely automatically.