Answering Queries with Aggregation Using Views

Divesh Srivastava
AT&T Research
divesh@research.att.com

Shaul Dar*
Data Technologies Ltd.
dar@dtl.co.il

H. V. Jagadish
AT&T Research
jag@research.att.com

Alon Y. Levy
AT&T Research
levy@research.att.com

Abstract

We present novel algorithms for the problem of using materialized views to compute answers to SQL queries with grouping and aggregation, in the presence of multiset tables. In addition to its obvious potential in query optimization, this problem is important in many applications, such as data warehousing, very large transaction recording systems, global information systems and mobile computing, where access to local or cached materialized views may be cheaper than access to the underlying database. Our contributions are the following: First, we show that in the case where the query has grouping and aggregation but the views do not, a view is usable in answering a query only if there is an isomorphism between the view and a portion of the query. Second, when the views also have grouping and aggregation we identify conditions under which the aggregation information present in a view is sufficient to perform the aggregation computations required in the query. The algorithms we describe for rewriting a query also consider the case in which the rewritten query may be a *union* of single-block queries. Our approach is a semantic one, in that it detects when the information existing in a view is sufficient to answer a query. In contrast, previous work performed syntactic transformations on the query such that the definition of the view would be a sub-part of the definition of the query. Consequently, these methods can only detect usages of views in limited cases.

1 Introduction

We present novel algorithms for the problem of using materialized views to compute answers to SQL queries with grouping and aggregation. This problem has the potential of improving the performance of SQL query evaluation in general. It has an even greater impact on the optimization of aggregation queries in applications such as data warehousing [GJM96, ZGMHW95], very large transaction recording systems [JMS95], global information systems [LSK95, LRO96] and mobile computing [BI94], where access to (local or cached) materialized views may be cheaper than access to the underlying database.

In data warehousing applications and very large transaction recording systems, the size of the database and the volume of incoming data may be very large. Queries against such data typically involve aggregation. Such queries may be answered more efficiently by materializing and maintaining appropriately defined aggregation views (summary tables), which are much smaller then the underlying data and can be cached in faster memory.

In globally distributed information systems, the relations may be distributed or replicated, and locating as well as accessing them may be expensive and sometimes not even possible. In mobile computing applications, the relations may be stored on a server and be accessible only via low bandwidth wireless communication, which may additionally become unavailable. Locally cached materialized views of the data, such as results of previous queries, may considerably improve the performance of such applications.

We formalize the problem of using materialized SQL views to answer SQL queries as finding a *rewriting* of a query Q where the views occur in the FROM clause, and the rewritten query is multiset-equivalent to Q. The technical challenges arise from the multiset semantics of SQL, in conjunction with the use of grouping and aggregation.

We focus on queries and views of the form "SELECT-FROM-WHERE-GROUPBY-HAVING", i.e., single-block queries, where the SELECT and HAVING clauses may contain the SQL aggregate functions MIN, MAX, SUM and COUNT.[1] We do not assume the availability of any meta-information about the schema, such as keys or functional dependencies. The contributions of this paper are developed in a step-wise fashion, as follows.

First, in Section 3, we study the case where the query has grouping and aggregation but the views do not. We consider rewritings that result in single-block queries, as well as rewritings that result in the UNION ALL (i.e., additive

*The work of this author was performed when he was at AT&T Bell Laboratories, Murray Hill, NJ, USA.

Proceedings of the 22nd VLDB Conference
Mumbai(Bombay), India, 1996

[1] The SQL aggregate functions SUM, COUNT and AVG are related in that, given values for two of them over some column, the third can be computed. Dealing with AVG is consequently straightforward, but complicates the presentation. Hence, we do not consider AVG.

multiset union) of single-block queries. We show that, for both types of rewritings, usability of a view in evaluating a query essentially requires an isomorphism between the view and a portion of the query: the view should not project out any column needed by the query, and it should retain all the tuples needed to compute information for some (all, if a single-block rewriting is desired) of the groups in the query. The rewriting algorithms can be iteratively applied to incorporate multiple views, and we identify the conditions under which *all* possible rewritings are generated.

Second, in Section 4, we study the case where both the query and the views have grouping and aggregation. Additional subtleties arise because an aggregated column can be regarded as being partially projected out, and a groupby in the view results in the multiplicities of the tuples being lost. We extend the conditions for usability in a natural fashion to recognize when the aggregation information present in a view is sufficient to perform the aggregate computations required in the query, and provide a rewriting algorithm for the query.

Finally, in Section 5, we show that when the views have grouping and aggregation but the query does not, it is not possible to use the views to evaluate the query. Intuitively, the loss of tuple multiplicities because of a groupby in the view prevents any multiset-equivalent rewriting.

There has been previous work on using views to answer queries (e.g., [YL87, SJGP90, TSI94, CR94, CKPS95, LMSS95]), but the problem of finding the equivalent rewritings for SQL queries with multiset semantics, grouping and aggregation, has received little attention. Several researchers have considered performing syntactic transformations on queries with grouping and aggregation that preserve equivalence of the query (e.g., [YL94, LMS94, CS94, RSSS95, GHQ95, CS96, LM96]). Gupta et al. [GHQ95] have shown how these transformations can be used for finding rewritings of queries by transforming the query in a way that the definition of the view would be identical to a sub-part of the definition of the query. In addition to being more restrictive than our semantic approach, the approach of Gupta et al. does not consider rewritings that are UNION ALLs of single-block queries. Hence, their approach can detect usages of views in only limited cases. A detailed comparison with related work is presented in Section 6.

1.1 Illustrative Example

We present an example from data warehousing in telephony to illustrate the potential performance gains when using materialized aggregation views to answer queries.

Example 1.1 Consider a data warehouse that holds information useful to a telephone company. The database maintains the following tables:

- $Customer(Phone\_Number, Cust\_Name)$, which maintains information about individual customers of the telephone company,
- $Calling\_Plans(Plan\_Id, Plan\_Name)$, which maintains information about the different calling plans of the telephone company, and
- $Calls(From, To, Time, Day, Month, Year, Duration, Plan\_Id, Charge)$, which maintains information about each individual call.

Assume that the telephone company is interested in determining calling plans that have earned more than a million dollars in one of the years between 1990 and 1995. The following SQL query Q_1 may be used for this purpose:

Q_1: SELECT $Year, Plan\_Name$, SUM($Charge$)
FROM $Calls, Calling\_Plans$
WHERE $Calls.Plan\_Id = Calling\_Plans.Plan\_Id$
AND $Year \geq 1990$ AND $Year \leq 1995$
GROUPBY $Year, Plan\_Name$
HAVING SUM($Charge$) $> 1,000,000$

The telephone company also maintains materialized views that summarize the performance of each of their calling plans on a periodical basis. In particular assume that the following materialized view $V_1(Plan\_Id, Month, Year, Earnings)$ is available:

V_1: SELECT $Plan\_Id, Month, Year$, SUM($Charge$)
FROM $Calls$
GROUPBY $Plan\_Id, Month, Year$

View V_1 can be used to evaluate the query Q_1 by joining V_1 with the table $Calling\_Plans$, collapsing multiple groups corresponding to the monthly plan earnings into annual plan earnings, and enforcing the additional conditions to get the summaries of plans earning more than a million dollars in one of the years between 1990 and 1995. The rewritten query Q_1' that uses V_1 is:

Q_1': SELECT $Year, Plan\_Name$, SUM($Earnings$)
FROM $V_1, Calling\_Plans$
WHERE $V_1.Plan\_Id = Calling\_Plans.Plan\_Id$
AND $Year \geq 1990$ AND $Year \leq 1995$
GROUPBY $Year, Plan\_Name$
HAVING SUM($Earnings$) $> 1,000,000$

The $Calls$ table may be huge, and the materialized view V_1 is likely to be orders of magnitude smaller than the $Calls$ table. Hence, evaluating Q_1' will be much faster than evaluating Q_1, emphasizing the importance of recognizing that Q_1 can be rewritten to use the materialized view V_1.

Consider now the case where, instead of V_1, the telephone company maintains the materialized view $V_1'(Plan\_Id, Month, Year, Earnings)$, summarizing the performance of their calling plans only since 1991:

V_1': SELECT $Plan\_Id, Month, Year$, SUM($Charge$)
FROM $Calls$
WHERE $Year \geq 1991$
GROUPBY $Plan\_Id, Month, Year$

View V_1' can still be used to evaluate query Q_1. However, not all the tuples in Q_1 can be computed using V_1'; the summary information computation for 1990 would have to access the $Calls$ table, and the rewritten query Q_1'' involves a UNION ALL.

Q_1'': SELECT $Year, Plan\_Name$, SUM($Earnings$)
FROM $V_1', Calling\_Plans$
WHERE $V_1'.Plan\_Id = Calling\_Plans.Plan\_Id$
AND $Year \leq 1995$
GROUPBY $Year, Plan\_Name$
HAVING SUM($Earnings$) $> 1,000,000$
UNION ALL
SELECT $Year, Plan\_Name$, SUM($Charge$)
FROM $Calls, Calling\_Plans$
WHERE $Calls.Plan\_Id = Calling\_Plans.Plan\_Id$
AND $Year = 1990$
GROUPBY $Year, Plan\_Name$
HAVING SUM($Charge$) $> 1,000,000$

Evaluating Q_1'' will still be faster than evaluating Q_1, even though it involves accessing the $Calls$ table. □

2 Notation and Definitions

We consider SQL queries and views with grouping and aggregation. Queries can be either *single-block* queries (described below), or union *multi-block* queries that are the UNION ALL (i.e., additive multiset union) of single-block queries. A view is defined by a query, and the name of the view is associated with the result of the query; in this paper, we consider only views defined by single-block queries. We give the form as well as a simple example of a single-block query in Figure 1.

For notational convenience, we modify the naming convention of standard SQL to guarantee *unique column names* for each of the columns in a single-block query. For example, let R_1 and R_2 be two tables each with a single column named A. If a single-block query Q has both R_1 and R_2 in its FROM clause, our notation would replace them by $R_1(A_1)$ and $R_2(A_2)$. Every reference to $R_1.A$ in Q is replaced by A_1, and every reference to $R_2.A$ in Q is replaced by A_2. Similarly, if a single-block query Q has two range variables R_1 and R_2 ranging over table R in its FROM clause, our notation would replace them by $R(A_1)$ and $R(A_2)$. Every reference to $R_1.A$ in Q is replaced by A_1, and every reference to $R_2.A$ in Q is replaced by A_2.

We use $Tables(Q)$ to denote the set of tables (along with their columns) $\{R_1(\bar{A}_1), \ldots, R_n(\bar{A}_n)\}$ in the FROM clause of a single-block query Q, and $Cols(Q)$ to denote $\bar{A}_1 \cup \ldots \cup \bar{A}_n$, i.e., the set of columns of tables in $Tables(Q)$. In the example of query Q_e, $Tables(Q_e)$ is $\{R(A, B), S(C, D, E)\}$ and $Cols(Q_e)$ is $\{A, B, C, D, E\}$.

The set of columns in the SELECT clause of Q, denoted by $Sel(Q)$, consists of: (a) *non-aggregation columns*: this is a subset of the columns in $Cols(Q)$, and is denoted by $ColSel(Q)$; and (b) *aggregation columns*: these are of the form **AGG**(Y), where Y is in $Cols(Q)$ and **AGG** is one of the aggregate functions MIN, MAX, SUM and COUNT. The set of columns that are aggregated upon, such as Y above, is a subset of $Cols(Q)$, and is denoted by $AggSel(Q)$. In the example of query Q_e, $Sel(Q_e)$ is $\{A, \text{MAX}(D), \text{SUM}(E)\}$, $ColSel(Q_e)$ is $\{A\}$ and $AggSel(Q_e)$ is $\{D, E\}$.

The grouping columns of query Q, denoted by $Groups(Q)$, consists of a subset of the columns in $Cols(Q)$. SQL requires that if $Groups(Q)$ is not empty, then $ColSel(Q)$ must be a subset of $Groups(Q)$. In the example of query Q_e, $Groups(Q_e)$ is $\{A, B\}$ and $ColSel(Q_e)$ is a proper subset of $Groups(Q_e)$.

We consider *built-in predicates* that are arithmetic predicates of the form α *op* β, where *op* is one of the comparison predicates $\{<, \leq, =, \neq, \geq, >\}$, and α and β are terms formed from columns of tables, aggregation columns and constants using the arithmetic operations $+$ and $*$.

The conditions in the WHERE clause of query Q, denoted by $Conds(Q)$, consists of a boolean combination of built-in predicates, formed using columns in $Cols(Q)$ and constants. The conditions in the HAVING clause of query Q, denoted by $GConds(Q)$, consists of a boolean combination of built-in predicates formed using columns in $Groups(Q)$, aggregation columns of the form **AGG**(Y) where Y is in $Cols(Q)$, and constants. In the example of query Q_e, $Conds(Q_e)$ is $B = C$, and $GConds(Q_e)$ is $\text{SUM}(D) > 1000$.

Given a single-block query Q, if $Groups(Q)$, $AggSel(Q)$ and $GConds(Q)$ are empty[2], then Q is referred to as a *conjunctive query*. Otherwise, Q is referred to as an *aggregation query*.

Determining that a single-block view V is usable in evaluating a single-block query Q requires (as we show later in the paper) that we consider mappings from V to Q. These are specified by column mappings, defined below.

Definition 2.1 (Column Mapping) A *column mapping* from a single-block query Q_a to a single-block query Q_b is a mapping ϕ from $Cols(Q_a)$ to $Cols(Q_b)$ such that if $R(A_1, \ldots, A_n)$ is a table in $Tables(Q_a)$, then: (1) there exists a table $R(B_1, \ldots, B_n)$ in $Tables(Q_b)$, and (2) $B_i = \phi(A_i), 1 \leq i \leq n$.

A *1-1 column mapping* ϕ is a column mapping from Q_a to Q_b such that distinct columns in $Cols(Q_a)$ are mapped to distinct columns in $Cols(Q_b)$. Otherwise, the column mapping is a *many-to-1 column mapping*. □

As a shorthand, if R is a table in $Tables(Q_a)$, we use $\phi(R(A_1, \ldots, A_n))$ to denote $R(\phi(A_1), \ldots, \phi(A_n))$, where

[2]Note that each of $Groups(Q)$, $AggSel(Q)$ and $GConds(Q)$ can be empty without the other two being empty.

Q: SELECT	$Sel(Q)$	Q_e : SELECT	A, MAX(D), SUM(E)
FROM	$R_1(\bar{A}_1), \ldots, R_n(\bar{A}_n)$	FROM	$R(A, B), S(C, D, E)$
WHERE	$Conds(Q)$	WHERE	$B = C$
GROUPBY	$Groups(Q)$	GROUPBY	A, B
HAVING	$GConds(Q)$	HAVING	SUM(D) > 1000

Figure 1: Form and example of a single-block query

$A_1, \ldots, A_n$ are columns in $Cols(Q_a)$. We use similar shorthand notation for mapping query results, sets and lists of columns, sets of tables, and conditions.

We formalize the intuitive notion of "usability" of view V in evaluating query Q as finding a *rewriting* of Q, defined below. In this paper, we consider only rewritings that are either single-block queries, or multi-block queries that are UNION ALLs of single-block queries. For example, rewriting Q'_1 in Example 1.1 is a single-block query, whereas rewriting Q''_1 in the same example is a multi-block query that is a UNION ALL of single-block queries.

Definition 2.2 (Rewriting of a query) A query Q' is a *rewriting* of query Q that uses view V if: (1) Q and Q' are multiset-equivalent, i.e., they compute the same *multiset* of answers for any given database, and (2) Q' contains one or more occurrences of V in the FROM clause of one of its blocks. □

In the sequel, we say that view V is *usable in evaluating query* Q, if there exists a single-block or a union multi-block query Q' such that Q' is a rewriting of Q that uses V.

When the rewritten query can be a multi-block query, there is a certain trivial sense in which *any* view V is usable in evaluating a given query Q — the rewritten query can be the UNION ALL of Q itself and a single-block query in which V occurs in the FROM clause and which has an unsatisfiable conjunction of built-in predicates in the WHERE clause. However, when Q is unsatisfiable, any rewriting of Q would also have to be unsatisfiable. Dealing with these and other such possibilities would complicate our presentation without aiding our understanding of the problem. Hence, we consider satisfiable queries and views, and do not permit multi-block rewritings where any block is unsatisfiable.

3 Aggregation Query and Conjunctive Views

In this section we consider the problem of using single-block conjunctive views to evaluate a single-block query with grouping and aggregation. Using a single-block view to evaluate a multi-block query can be achieved by independently testing usability of the view in evaluating each block of the multi-block query separately.

Intuitively, if a view V is usable in evaluating a query Q, then V must "replace" some of the tables and conditions enforced in Q; other tables and conditions from Q must remain in the rewritten query Q'. The rewritten query Q' can be a single-block query, or a multi-block query that is a UNION ALL of single-block queries. For view V to be usable in answering query Q, such that Q' is a single-block query, it must be the case that:

- V does not project out any *columns* needed by Q.

  Intuitively, a column A is needed by Q if it appears in the result of Q or if Q needs to enforce a condition involving A that has *not* been enforced in the computation of V.

- V does not discard any *tuples* needed by Q.

  Intuitively, a tuple is needed by Q if it satisfies the conditions enforced in Q.

When Q' can be a multi-block query, the second requirement can be somewhat relaxed to require that V not discard any tuples needed for some of the groups in Q.

We formalize these intuitions below, show that they yield both necessary and sufficient conditions for certain kinds of queries, and present an algorithm to rewrite Q using V. We first examine the case when the query does not have a HAVING clause, and then describe the effect of the HAVING clause on the conditions for usability and the rewriting algorithm.

3.1 Aggregation Query Without a HAVING Clause

3.1.1 Single-Block Rewritten Query

The conditions for usability of a single-block view V in evaluating a single-block query Q, such that the rewritten query Q' is a single-block query, are presented formally in Figure 2 in terms of column mappings. Note that the conditions apply also to the restricted case when both the view and the query are conjunctive [CKPS95].

Condition C_1 and the first part of condition C_4 essentially guarantee that the view is multiset equivalent to its image under ϕ; these are a reformulation of the conditions presented in [CV93] for testing equivalence of conjunctive queries under the multiset semantics. Note that the 1-1 mapping is necessary because of the multiset semantics, whereas a many-to-1 mapping would suffice in the case of sets [LMSS95]. Condition C_4 ensures that constraints not enforced in the view can still be enforced in the query when the view is used, since they do not refer to columns that are projected out in the view and hence are no longer available. Conditions C_2 and C_3 ensure that the view does not project

Condition C_1 : There is a 1-1 column mapping ϕ from V to Q.

Condition C_2 : If a column A in $ColSel(Q) \cup Groups(Q)$ is a column in $\phi(Cols(V))$, then $Sel(V)$ must have a column B_A such that $Conds(Q)$ implies $(A = \phi(B_A))$.

Note that this condition is satisfied if B_A is $\phi^{-1}(A)$.

Condition C_3 : Suppose **AGG**(A) is in $Sel(Q)$. If column A is in $\phi(Cols(V))$, then:

1. If **AGG** is MIN, MAX or SUM, then $Sel(V)$ must have a column B_A such that $Conds(Q)$ implies $(A = \phi(B_A))$.
2. If **AGG** is COUNT, then $Sel(V)$ must not be empty.

Condition C_4 : There exists a boolean combination of built-in predicates, $Conds'$, such that:

1. $Conds(Q)$ is equivalent to $\phi(Conds(V))$ & $Conds'$.
2. $Conds'$ involves only the columns in $\phi(Sel(V)) \cup (Cols(Q) - \phi(Cols(V)))$.

Figure 2: Usability conditions for a single-block aggregation query without a HAVING clause, a single-block conjunctive view, with a single-block rewritten query

out any column that is required in the SELECT clause of the query. Condition C_3 is the one needed in order to deal with the aggregation in the query.

If conditions $C_1 - C_4$ are satisfied, the rewritten query Q' is obtained from Q by replacing the tables in $\phi(Tables(V))$ by $\phi(V)$ in the FROM clause, where $\phi(V)$ denotes $V(\phi(Sel(V)))$. The SELECT and WHERE clauses of the query are then modified to reflect the use of view V in the rewritten query. Formally, the single-block rewritten query Q' is obtained from Q by applying algorithm **ConjViewSingleBlock**, presented in Figure 3.

Theorem 3.1 *Let Q be a single-block aggregation query without a HAVING clause, and let V be a single-block conjunctive view.*

If conditions $C_1 - C_4$ are satisfied, V is usable in evaluating Q. In that case Q', obtained by applying algorithm **ConjViewSingleBlock**, *is a rewriting of Q using V.*

If $Conds(Q)$ and $Conds(V)$ contain only equality predicates of the form $A = B$, where A and B are column names or constants, and the rewritten query is required to be a single-block query, V is usable in evaluating Q only if conditions $C_1 - C_4$ are satisfied. □

The following example illustrates conditions $C_1 - C_4$ and algorithm **ConjViewSingleBlock** for obtaining a single-block rewritten query.

Example 3.1 Consider the telephone company database from Example 1.1. The following query Q_2 can be used to determine the total earnings of each calling plan as well as the total number of calls charged under each calling plan in December 1995.

Algorithm ConjViewSingleBlock

Step S_1 : Replace all the tables in $\phi(Tables(V))$ by $\phi(V)$.

Step S_2 : Replace each column A in $Groups(Q) \cup ColSel(Q) \cup AggSel(Q)$ by $\phi(B_A)$, where B_A satisfies conditions C_2 and C_3, part 1.

Step S_3 : Determine a boolean combination of built-in predicates $Conds'$ satisfying condition C_4. Replace $Conds(Q)$ in Q by $Conds'$.

Step S_4 : Consider an aggregation column COUNT(A) in $Sel(Q)$ such that A is in $\phi(Cols(V))$, but not in $\phi(Sel(V))$. Replace COUNT(A) by COUNT(B), where B is any column in $\phi(V)$.

Figure 3: Rewriting algorithm for a single-block aggregation query without a HAVING clause, a single-block conjunctive view, with a single-block rewritten query

Q_2: SELECT PN_1, SUM(C_1), COUNT(C_1)
FROM $Calls(F_1, T_1, TI_1, D_1, M_1, Y_1, DU_1, P_1, C_1)$, $Calling\_Plans(PI_1, PN_1)$
WHERE $P_1 = PI_1$ AND $Y_1 = 1995$ AND $M_1 = 12$
GROUPBY PN_1

Assume that the telephone company maintains call data for December 1995 as the view V_2 below:

V_2: SELECT $F_2, T_2, TI_2, D_2, M_2, Y_2, DU_2, P_2, C_2$
FROM $Calls(F_2, T_2, TI_2, D_2, M_2, Y_2, DU_2, P_2, C_2)$
WHERE $Y_2 = 1995$ AND $M_2 = 12$

View V_2 can be used to evaluate query Q_2 since conditions $C_1 - C_4$ are satisfied: (C_1) The 1-1 column mapping ϕ from V_2 to Q_2 is $\{F_2 \rightarrow F_1, T_2 \rightarrow T_1, TI_2 \rightarrow TI_1, D_2 \rightarrow D_1, M_2 \rightarrow M_1, Y_2 \rightarrow Y_1, DU_2 \rightarrow DU_1, P_2 \rightarrow P_1, C_2 \rightarrow C_1\}$. ($C_2$) Trivially satisfied. ($C_3$) For column C_1, B_{C_1} is the column C_2 in $Sel(V_2)$. (C_4) $Conds'$ is given by $P_1 = PI_1$.

The single-block rewriting of Q_2 that uses V_2 is:

Q_2': SELECT PN_1, SUM(C_1), COUNT(C_1)
FROM $V_2(F_1, T_1, TI_1, D_1, M_1, Y_1, DU_1, P_1, C_1)$, $Calling\_Plans(PI_1, PN_1)$
WHERE $P_1 = PI_1$
GROUPBY PN_1

□

3.1.2 Multi-Block Rewritten Query

When the rewritten query is not required to be a single-block query, but can be a multi-block query that is a UNION ALL of single-block queries, additional usages of views in evaluating queries are possible.

Condition C_4^m : Let $Conds_1$ be $Conds(Q)$ & $\phi(Conds(V))$, and $Conds_2$ be $Conds(Q)$ & $\neg\phi(Conds(V))$. Then,

1. $Conds_1$ must be satisfiable.
2. There exists a boolean combination of built-in predicates, $Conds'$, such that:
   (a) $Conds_1$ is equivalent to $\phi(Conds(V))$ & $Conds'$,
   (b) $Conds'$ involves only the columns in $\phi(Sel(V)) \cup (Cols(Q) - \phi(Cols(V)))$.
   (c) $\pi_{Groups(Q)}(Conds_1)$ & $\pi_{Groups(Q)}(Conds_2)$ is FALSE.

Figure 4: Modification of condition C_4, when multi-block rewritten queries are permitted

The conditions for usability of a single-block view V in evaluating a single-block query Q, when Q' can be a multi-block rewritten query, are similar to the conditions for usability when Q' has to be a single-block query. In particular, conditions $C_1 - C_3$ are unchanged. Condition C_4 has to be modified to reflect the possibility that V can be used to compute only some of the tuples of Q. The modified condition, C_4^m, is formally presented in Figure 4.

Intuitively, given a view V that satisfies condition C_1, query Q can always be reformulated as a UNION ALL of 2 single-block queries Q_a and Q_b, that differ from Q (and from each other) only in their WHERE clauses, such that: (1) $Conds(Q_a)$ is equivalent to $Conds(Q)$ & $\phi(Conds(V))$, and (2) $Conds(Q_b)$ is equivalent to $Conds(Q)$ & $\neg\phi(Conds(V))$.

View V can be potentially used to evaluate Q_a, but clearly not Q_b. Conditions $C_1 - C_3$, and parts 1, 2(a) and 2(b) of condition C_4^m essentially check whether view V can be used to evaluate Q_a. The reformulation of Q as the UNION ALL of Q_a and Q_b, however, does not always preserve the semantics of Q. To preserve the semantics, it must be guaranteed that Q_a and Q_b do not compute tuples for the *same group* of Q – part 2(c) of condition C_4^m embodies this requirement. If conditions $C_1 - C_3$ and C_4^m are satisfied, the multi-block rewritten query Q' is obtained, using algorithm **ConjViewMultiBlock**, presented in Figure 5.

Theorem 3.2 *Let Q be a single-block aggregation query without a HAVING clause, and let V be a single-block conjunctive view.*

If conditions $C_1 - C_3$ and C_4^m are satisfied, V is usable in evaluating Q. In that case Q', obtained by applying algorithm **ConjViewMultiBlock**, *is a multi-block rewriting of Q using V.* □

3.2 Multiple Uses of Views

Often a query can make use of multiple views, or the same view multiple times. The rewriting algorithms **ConjViewSingleBlock** and **ConjViewMultiBlock** presented above can be used to incorporate multiple uses of views. To obtain rewritings with multiple views we create successive rewritings $Q'_a, \ldots, Q'_n$ where each rewriting is obtained from the previous one by testing conditions $C_1 - C_3$ and either C_4 or C_4^m (depending on the form of the rewriting desired), and applying the corresponding rewriting algorithm. At each successive rewriting, the views incorporated in previous rewritings are treated as database tables, rather than being expanded using their view definitions.

Algorithm **ConjViewMultiBlock**

Step S_1^m : Use $\phi(Conds(V))$ to split Q into Q_a and Q_b, such that $Conds(Q_a)$ is equivalent to $Conds(Q)$ & $\phi(Conds(V))$, and $Conds(Q_b)$ is equivalent to $Conds(Q)$ & $\neg\phi(Conds(V))$.

Step S_2^m : Use algorithm **ConjViewSingleBlock** to rewrite Q_a to make use of view V. Let Q'_a denote the resultant single-block query.

Step S_3^m : If Q_b is satisfiable, the multi-block query Q' that is the rewriting of Q using V is the UNION ALL of Q'_a and Q_b. Else, Q' is the same as Q'_a.

Figure 5: Rewriting algorithm for a single-block aggregation query without a HAVING clause, a single-block conjunctive view, with a multi-block rewritten query

Theorem 3.3 *Let Q be a single-block aggregation query without a HAVING clause, and let $V_a, \ldots, V_m$ be single-block conjunctive views. Then the following hold:*

1. *An iterative application of algorithm* **ConjViewSingleBlock** *is* sound, *i.e., each successive rewriting is multiset-equivalent to Q.*
2. *An iterative application of algorithm* **ConjViewMultiBlock** *is* sound, *i.e., each successive rewriting is multiset-equivalent to Q.*
3. *The rewriting algorithm* **ConjViewSingleBlock** *is* order-independent. *That is, if there is a single-block rewriting of Q that uses each of $V_a, \ldots, V_m$, then the result of rewriting Q to incorporate views $V_a, \ldots, V_m$ would be the same regardless of the order in which the views are considered.*
4. *If $Conds(Q), Conds(V_a), \ldots, Conds(V_m)$ contain only equality predicates of the form $A = B$, where A and B are column names, or constants, and the rewritten query is required to be a single-block query, then the iterative application of algorithm* **ConjViewSingleBlock** *is* complete. *That is, any rewriting of Q that uses one or more of $V_a, \ldots, V_m$ can be obtained by iteratively applying algorithm* **ConjViewSingleBlock**.

□

Condition C_3^h : Suppose **AGG(A)** is in $Sel(Q)$ or in $GConds(Q)$. If column A is in $\phi(Cols(V))$, then:

1. If **AGG** is MIN, MAX or SUM, then $Sel(V)$ must have a column B_A such that $Conds(Q)$ implies $(A = \phi(B_A))$.
2. If **AGG** is COUNT, then $Sel(V)$ must not be empty.

Figure 6: Modification of condition C_3, when query Q has a HAVING clause

It is important to note that, for the case of equality predicates, the iterative application of **ConjViewSingleBlock** guarantees that we find *all* ways of using the views to answer a query, provided the rewritten query is required to be a single-block query. In contrast, this property does not hold under the set semantics considered in [LMSS95], where there may exist rewritings that cannot be found by considering sequences of single view substitutions.

3.3 Aggregation Query With a HAVING Clause

We now describe how to extend the previous algorithms to the case in which the queries may contain a HAVING clause. We only consider the case when the rewritten query is required to be a single-block query. The case when the rewritten query can be a multi-block query is a straightforward extension, along the lines described for aggregation queries without HAVING clauses. We first describe how to extend our usability conditions to accommodate the HAVING clause, and then show how we can use various transformations on the query that can cause the conditions to be satisfied in a larger number of cases.

Intuitively, when the single-block query Q has a HAVING clause, the conditions for usability of a conjunctive view V in evaluating Q and the rewriting algorithm **ConjViewSingleBlock** need to be extended to account for:

- Conditions in $GConds(Q)$ that must be satisfied by the query, in addition to conditions in $Conds(Q)$, and
- Aggregation columns, of the form **AGG(Y)**, that occur in $GConds(Q)$, but not in $Sel(Q)$.

To accommodate such conditions we modify C_3 to also consider arguments that appear in $GConds(Q)$. The extended condition, C_3^h, is formally presented in Figure 6. If Q and V satisfy conditions C_1, C_2, C_3^h and C_4, the single-block rewritten query Q' is obtained using algorithm **HavingConjViewSingleBlock**, presented in Figure 7.

Theorem 3.4 *Let Q be a single-block aggregation query with a HAVING clause, and let V be a single-block conjunctive view.*

If conditions C_1, C_2, C_3^h and C_4 are satisfied, V is usable in evaluating Q. In that case Q', obtained by applying algorithm **HavingConjViewSingleBlock**, *is a rewriting of Q using V.* □

Algorithm **HavingConjViewSingleBlock**
Assume that the query Q has been pre-processed.

Step S_1^h : Apply steps S_1, S_2 and S_3 using condition C_3^h instead of condition C_3.

Step S_2^h : Replace each column A in $GConds(Q)$ by $\phi(B_A)$, where A and B_A satisfy conditions C_2 and C_3^h, part 1.

Step S_3^h : Consider an aggregation column COUNT(A) in $Sel(Q)$ or in $GConds(Q)$ such that A is in $\phi(Cols(V))$, but not in $\phi(Sel(V))$. Replace COUNT(A) by COUNT(B), where B is any column in $\phi(V)$.

Figure 7: Rewriting algorithm for a single-block aggregation query with a HAVING clause, a single-block conjunctive view, with a single-block rewritten query

Strengthening the Conditions in the Query

When query Q has a HAVING clause, the conditions in its HAVING clause may enable us to *strengthen* the conditions in its WHERE clause, without affecting the result of the query. Strengthening the conditions in the WHERE clause may allow us to detect usability of views that would otherwise not be determined to be usable, because it makes it more likely that condition C_4 will be satisfied.

Several authors (e.g., [LMS94, RSSS95, GHQ95, LM96]) have considered the problem of inferring conditions that can be conjoined to $Conds(Q)$ given the conditions in $GConds(Q)$, and removing redundant conditions in $GConds(Q)$. These techniques can be applied to rewrite the query Q, as a pre-processing step, yielding possibly modified conditions $Conds(Q)$ and $GConds(Q)$. The modified $Conds(Q)$ and $GConds(Q)$ are then used in checking conditions C_2, C_3^h and C_4.

Example 3.2 Consider again the telephone company database from Example 1.1. The following query Q_3 can be used to determine, for each customer, the maximum charge for a single call under the calling plan "TrueUniverse" in December 1995, provided that the charge exceeds \$10.

Q_3: SELECT $F_1, MAX(C_1)$
FROM $Calls(F_1, T_1, TI_1, D_1, M_1, Y_1, DU_1, P_1, C_1)$, $Calling\_Plans(PI_1, PN_1)$
WHERE $P_1 = PI_1$ AND PN_1 = "TrueUniverse" AND $Y_1 = 1995$ AND $M_1 = 12$
GROUPBY F_1
HAVING $MAX(C_1) > 10$

Assume that the telephone company maintains detailed call data for 1995, for calls whose charge exceeds \$1, as the view V_3 below:

V_3: SELECT $F_2, T_2, TI_2, D_2, M_2, Y_2, DU_2, P_2, C_2$
FROM $Calls(F_2, T_2, TI_2, D_2, M_2, Y_2, DU_2, P_2, C_2)$
WHERE $Y_2 = 1995$ AND $C_2 > 1$

Although the WHERE clause of Q_3 does not enforce any conditions on the *Charge* column, while the WHERE clause of V_3 does, V_3 can still be used to evaluate Q_3. This is because the condition MAX(C_1) > 10 in the HAVING clause of Q_3 is equivalent to having the condition $C_1 > 10$ in the WHERE clause of Q_3. Strengthening $Conds(Q_3)$ by conjoining $C_1 > 10$ (and subsequently removing the redundant HAVING clause) allows the detection of usability of V_3 in evaluating Q_3. The rewriting of Q_3 that uses V_3 is:

Q_3': SELECT F_1, MAX(C_1)
FROM $V_3(F_1, T_1, TI_1, D_1, M_1, Y_1, DU_1, P_1, C_1)$, $Calling\_Plans(PI_1, PN_1)$
WHERE $P_1 = PI_1$ AND PN_1 = "TrueUniverse" AND $M_1 = 12$ AND $C_1 > 10$
GROUPBY F_1

Note that view V_3 cannot be used to answer query Q_2 (from Example 3.1), since conditions C_4 and C_4^m are violated — in particular, V_3 enforces the condition $C_2 > 1$, which results in the discarding of *Calls* tuples needed by Q_2. □

4 Aggregation Query and Views

In this section we consider the problem of using single-block views in evaluating single-block queries when both the view and the query have grouping and aggregation. We only consider the case when the rewritten query is required to be a single-block query.

Recall that the two intuitive requirements for the usability of a conjunctive view V in answering a single-block aggregation query Q (described in the beginning of Section 3) are that V not project out columns needed in Q, and that V not discard tuples needed in Q. In the presence of grouping and aggregation in the view, these requirements become more subtle:

- An aggregation over a column in V can be thought of as though that column was *partially* projected out, since V contains just aggregate values over that column, not the original column values themselves.
- A groupby in V results in the multiplicities of the tuples being lost.

However, as the following examples illustrate, in some cases it is possible to overcome the difficulties introduced by grouping and aggregation in the view.

4.1 Illustrative Examples

The following example illustrates that the aggregate information in a view may be sufficient to compute the aggregate information needed in the query.

Example 4.1 (Coalescing Subgroups)
Consider the telephone company database from Example 1.1. The following query Q_4 can be used to determine the total earnings of various calling plans as well as the maximum charge under each calling plan in 1995.

Q_4: SELECT P_1, PN_1, SUM(C_1), MAX(C_1)
FROM $Calls(F_1, T_1, TI_1, D_1, M_1, Y_1, DU_1, P_1, C_1)$, $Calling\_Plans(PI_1, PN_1)$
WHERE $P_1 = PI_1$ AND $Y_1 = 1995$
GROUPBY P_1, PN_1

Assume that the telephone company also maintains information giving the total earnings as well as the maximum charge of each calling plan in each month in the form of view V_4 below:

V_4: SELECT P_2, M_2, Y_2, SUM(C_2), MAX(C_2)
FROM $Calls(F_2, T_2, TI_2, D_2, M_2, Y_2, DU_2, P_2, C_2)$
GROUPBY P_2, M_2, Y_2

View V_4 groups the table *Calls* by the *Plan_Id*, *Month* and *Year* columns, and computes aggregate information on each such group. Query Q_4, on the other hand, groups the table *Calls* only on the *Plan_Id* column, resulting in more coarse groups than those computed in V_4. However, the aggregate information of the *Plan_Id* groups in Q_4 can be computed by further aggregating the aggregate information computed for the (*Plan_Id*, *Month*, *Year*) groups in V_4, as illustrated in the following rewritten query.

Q_4': SELECT P_1, PN_1, SUM(ME_1), MAX(MC_1)
FROM $V_4(P_1, M_1, Y_1, ME_1, MC_1)$, $Calling\_Plans(PI_1, PN_1)$
WHERE $P_1 = PI_1$ AND $Y_1 = 1995$
GROUPBY P_1, PN_1

□

The following example illustrates that the existence of other columns in the view may enable us to recover the tuple multiplicities lost because of grouping in the view.

Example 4.2 (Recovery of Lost Multiplicities)
Consider again the telephone company database from Example 1.1. The following query Q_5 can be used to determine the total number of calls under each calling plan in 1995:

Q_5: SELECT P_1, COUNT(CN_1)
FROM $Calls(F_1, T_1, TI_1, D_1, M_1, Y_1, DU_1, P_1, C_1)$, $Customer(PN_1, CN_1)$
WHERE $F_1 = PN_1$ AND $Y_1 = 1995$
GROUPBY P_1

View V_{5a} below maintains the total annual revenue for each customer, plan, and year:

V_{5a}: SELECT F_2, P_2, Y_2, SUM(C_2)
FROM $Calls(F_2, T_2, TI_2, D_2, M_2, Y_2, DU_2, P_2, C_2)$
GROUPBY F_2, P_2, Y_2

Although V_{5a} does not project out any column that is needed in Q_5, V_{5a} *cannot* be used to evaluate Q_5. This is because the multiplicity of the $From$ column of $Calls$ is needed in order to compute COUNT(CN_1), but that multiplicity is lost in the view V_{5a}. However, consider view V_{5b} below:

V_{5b}: SELECT F_2, P_2, Y_2, SUM(C_2), COUNT(C_2)
FROM $Calls(F_2, T_2, TI_2, D_2, M_2, Y_2, DU_2, P_2, C_2)$
GROUPBY F_2, P_2, Y_2

Although the multiplicities of the $From$ column are not explicit in V_{5b}, they can be computed using the available information. V_{5b} can be used to evaluate Q_5 as follows:

Q_5': SELECT P_1, SUM(CG_1)
FROM $V_{5b}(F_1, P_1, Y_1, YE_1, CG_1)$, $Customer(PN_1, CN_1)$
WHERE $F_1 = PN_1$ AND $Y_1 = 1995$
GROUPBY P_1

□

As the examples illustrate, to use views that involve aggregations, we need to verify that (a) the aggregate information in the view is sufficient to compute the aggregates needed in the query, and that (b) the correct multiplicities exist or can be computed. We formalize these intuitions below, present conditions for usability, and provide an algorithm to rewrite Q using V.

4.2 Without HAVING Clauses

To specify conditions for usability for single-block aggregation views, we need to slightly modify conditions C_2 and C_4, and to substantially modify condition C_3 to deal with the different cases of aggregates appearing in the SELECT clause of the query. (Condition C_1 is unchanged.) The modified conditions are formally presented in Figure 8.

Since $ColSel(Q)$ must be a subset of $Groups(Q)$, condition C_2^a is a generalization of condition C_2. Intuitively, condition C_3^a guarantees that the columns in the view contain enough information to compute the aggregates required in the query. In particular, conditions C_3^a, parts 1(b), 1(c) and 2 guarantee that we can recover the multiplicities in the view in order to perform an aggregation that depends on such multiplicities (i.e., either SUM or COUNT). The two parts of the condition cover the cases when the aggregation is on a column mapped by the view, and not mapped by the view, respectively. Note that the second part of condition C_4^a does not allow $Conds'$ to constrain any of the columns in $\phi(AggSel(V))$. Intuitively, this is because the columns in $AggSel(V)$ are aggregated upon in view V, and hence are not "available" for imposition of additional constraints in the rewritten query Q'.

If conditions $C_1^a - C_4^a$ are satisfied, the rewritten query Q' is obtained from Q by applying algorithm **AggViewSingleBlock**, presented in Figure 9. Steps S_1^a, S_2^a and S_3^a are similar to steps S_1, S_2 and S_3 of algorithm **ConjViewSingleBlock**. Steps S_4^a and S_5^a deal with the various kinds of aggregation that may occur in the view and the query.

Theorem 4.1 *Let Q and V be single-block aggregation queries without HAVING clauses.*

If conditions $C_1^a - C_4^a$ are satisfied, V is usable in evaluating Q. In that case Q', obtained by applying algorithm **AggViewSingleBlock**, *is a rewriting of Q using V.* □

Example 4.3 Consider again the query Q_4 and view V_4 from Example 4.1. View V_4 can be used to evaluate Q_4 since conditions $C_1^a - C_4^a$ are satisfied.

Condition C_1^a : The 1-1 column mapping ϕ from V_4 to Q_4 is $\{F_2 \rightarrow F_1, T_2 \rightarrow T_1, TI_2 \rightarrow TI_1, D_2 \rightarrow D_1, M_2 \rightarrow M_1, Y_2 \rightarrow Y_1, DU_2 \rightarrow DU_1, P_2 \rightarrow P_1, C_2 \rightarrow C_1\}$.

Condition C_2^a : For column P_1 in $Groups(Q_4)$, B_{P_1} is the column P_2 in $ColSel(V_4)$.

Condition C_3^a : For column SUM(C_1) in $Sel(Q_4)$, $Sel(V_4)$ contains column SUM(C_2), and for column MAX(C_1) in $Sel(Q_4)$, $Sel(V_4)$ contains column MAX(C_2).

Condition C_4^a : $Conds'$ is the same as $Conds(Q_4)$, i.e., $P_1 = PI_1 \& Y_1 = 1995$ since no conditions are enforced in V_4.

The rewritten query Q_4' resulting from applying steps S_1^a – S_5^a is given in Example 4.1. □

Example 4.4 (Constraining $\phi(AggSel(V))$)
Consider again the telephone company database from Example 1.1. The following query Q_6 can be used to determine the total earnings of various calling plans in 1995, considering only calls whose charge exceeds \$1.

Q_6: SELECT P_1, SUM(C_1)
FROM $Calls(F_1, T_1, TI_1, D_1, M_1, Y_1, DU_1, P_1, C_1)$
WHERE $Y_1 = 1995$ AND $C_1 > 1$
GROUPBY P_1

Let the view V_6 be the same as view V_4 (from Example 4.1):

V_6: SELECT P_2, M_2, Y_2, SUM(C_2), MAX(C_2)
FROM $Calls(F_2, T_2, TI_2, D_2, M_2, Y_2, DU_2, P_2, C_2)$
GROUPBY P_2, M_2, Y_2

View V_6 cannot be used to evaluate Q_6 above, although in the absence of the condition "$C_1 > 1$" in the WHERE clause in Q_6, V_6 could be used to evaluate Q_6. Intuitively, this is because the built-in predicates in the query constrain the possible values of C_1, and C_2 is aggregated upon in the view V_6; no condition on the result of the SUM or the MAX in V_6 can capture the effect of the condition on C_1 in Q_6. □

Condition C_1^a : Same as condition C_1.

Condition C_2^a : If a column A in $Groups(Q)$ is a column in $\phi(Cols(V))$, then $ColSel(V)$ must have a column B_A such that $Conds(Q)$ implies $(A = \phi(B_A))$.

Condition C_3^a : Suppose **AGG**(A) is in $Sel(Q)$.

1. If column A is in $\phi(Cols(V))$, then:
   (a) If **AGG** is MIN or MAX, then there must exist a column B_A in $Cols(V)$ such that $Conds(Q)$ implies $(A = \phi(B_A))$, and $Sel(V)$ contains either the non-aggregation column B_A, or an aggregation column of the form **AGG**(B_A).
   (b) If **AGG** is COUNT, then $Sel(V)$ must include a column of the form COUNT(A_0), where A_0 is a column in $Cols(V)$.
   (c) If **AGG** is SUM, then there must exist a column B_A in $Cols(V)$ such that $Conds(Q)$ implies $(A = \phi(B_A))$, and $Sel(V)$ contains either B_A and a column of the form COUNT(A_0), or an aggregation column of the form **AGG**(B_A).
2. If column A is not in $\phi(Cols(V))$, and **AGG** is either SUM or COUNT, then $Sel(V)$ must include a column of the form $COUNT(A_0)$, where A_0 is a column in $Cols(V)$.

Condition C_4^a : There exists $Conds'$, such that:

1. $Conds(Q)$ is equivalent to $\phi(Conds(V))$ & $Conds'$.
2. $Conds'$ involves only the columns in $\phi(ColSel(V))$ and the columns in $Cols(Q)$ that are not in $\phi(Cols(V))$.

Figure 8: Usability conditions for a single-block aggregation query without a HAVING clause, a single-block aggregation view without a HAVING clause, with a single-block rewritten query

Algorithm **AggViewSingleBlock**

Step S_1^a : Replace all the tables in $\phi(Tables(V))$ by $\phi(V)$, where $\phi(V)$ is defined as follows: for each non-aggregation column A in $Sel(V)$, $\phi(V)$ contains the column $\phi(A)$; for each aggregation column A in $Sel(V)$, $\phi(V)$ contains a new column name.

Step S_2^a : Replace each column A in $Groups(Q) \cup ColSel(Q) \cup AggSel(Q)$ by $\phi(B_A)$, where B_A satisfies conditions C_2^a and C_3^a, part 1(a).

In the remaining steps of this algorithm, $Groups(Q)$, $ColSel(Q)$ and $AggSel(Q)$ refer to these new column names.

Step S_3^a : Determine a boolean combination of built-in predicates $Conds'$ satisfying condition C_4^a as above. Replace $Conds(Q)$ in Q by $Conds'$.

Step S_4^a : Consider an aggregation column **AGG**(A) in $Sel(Q)$ such that A is in $\phi(Cols(V))$.

1. Let **AGG** be MIN, MAX or SUM. By condition C_3^a, part 1, there are two cases to consider.
   (a) Suppose $Sel(V)$ contains the aggregation column **AGG**(B_A). Let S denote the corresponding column in $\phi(V)$. Replace **AGG**(A) in $Sel(Q)$ by **AGG**(S).
   (b) Suppose $Sel(V)$ contains the non-aggregation column B_A.
   If **AGG** is either MIN or MAX, leave **AGG**(A) in $Sel(Q)$ unchanged.
   If **AGG** is SUM, then by condition C_3^a, part 1(c), $Sel(V)$ must include a column of the form COUNT(A_0). Let N denote the corresponding column in $\phi(V)$. Replace SUM(A) in $Sel(Q)$ by SUM($A * N$).
2. Let **AGG** be COUNT. By condition C_3^a, part 1(b), $Sel(V)$ must include a column of the form COUNT(A_0). Let N denote the corresponding column in $\phi(V)$. Replace COUNT(A) in $Sel(Q)$ by SUM(N).

Step S_5^a : Consider an aggregation column **AGG**(A) in $Sel(Q)$ such that column A is not in $\phi(Cols(V))$.

If **AGG** is MIN or MAX, leave **AGG**(A) unchanged.

If **AGG** is SUM or COUNT, do the following. By condition C_3^a, part 2, $Sel(V)$ must include a column of the form COUNT(A_0). Let N denote the column in $\phi(V)$ corresponding to that COUNT(A_0) column.

1. If **AGG** is COUNT, replace COUNT(A) in $Sel(Q)$ by SUM(N).
2. If **AGG** is SUM, replace SUM(A) in $Sel(Q)$ by SUM($A * N$).

Figure 9: Rewriting algorithm for a single-block aggregation query without a HAVING clause, a single-block aggregation view without a HAVING clause, with a single-block rewritten query

4.3 With HAVING Clauses

Essentially, the additional subtleties that must be considered involve the relationships between the GROUPBY and HAVING clauses in the view V and the query Q. Intuitively, the HAVING clause in V may eliminate certain groups in V (i.e., those that do not satisfy $GConds(V)$). If any of these eliminated groups in V is "needed" to compute an aggregate function over a group in Q, by coalescing multiple groups in V, then V cannot be used to evaluate Q. Hence, condition C_4^a must be extended to test whether there exists $GConds'$ such that $GConds(Q)$ is equivalent to the combination of $GConds(V)$ and $GConds'$, taking the grouping columns $Groups(V)$ and $Groups(Q)$ into account.

Before checking any of the conditions for usability, the query Q and view V can be *independently* pre-processed to "move" maximal sets of conditions from the HAVING clause to the WHERE clause, as discussed in Section 3.3; the resulting normal form allows independent comparison of $Conds(Q)$ and $Conds(V)$, on the one hand, and of $GConds(Q)$ and $GConds(V)$, on the other.

The rewriting algorithm takes these additional refinements of the conditions of usability into account. Specifically, step S_3^a determines a $GConds'$ in addition to $Conds'$, using $GConds(V)$ and $GConds(Q)$ (resulting from the pre-processing step). Steps S_4^a and S_5^a are augmented to compute aggregation columns appearing in $GConds(Q)$, in addition to those appearing in $Sel(Q)$.

5 Conjunctive Query and Aggregation Views

Consider the case when the query Q is a conjunctive query (i.e., no grouping and aggregation), but the view V has grouping and aggregation. In this case, the GROUPBY clause in the view results in losing information about the multiplicities of tuples, and view V *cannot* be used to evaluate Q if the multiset semantics is desired.

Theorem 5.1 *Let Q be a conjunctive query, and V be a single-block aggregation view. Then, there is* no *single-block rewriting of Q using V.* □

The following example illustrates the problem with conjunctive queries and aggregation views:

Example 5.1 Consider the telephone company database from Example 1.1. The query Q_7 below is used to obtain information about calls exceeding an hour in duration:

Q_7: SELECT F_1, D_1, M_1, Y_1
FROM $Calls(F_1, T_1, TI_1, D_1, M_1, Y_1, DU_1, P_1, C_1)$
WHERE $DU_1 > 3600$

The view V_7 below counts the number of calls exceeding an hour in duration made by each caller on a daily basis:

V_7: SELECT F_2, D_2, M_2, Y_2, COUNT(T_2)
FROM $Calls(F_2, T_2, TI_2, D_2, M_2, Y_2, DU_2, P_2, C_2)$
WHERE $DU_2 > 3600$
GROUPBY F_2, D_2, M_2, Y_2

There is a 1-1 column mapping from V_7 to Q_7, $Sel(V_7)$ contains all the columns required in $Sel(Q_7)$, and the conditions enforced by the WHERE clauses are identical. Even though COUNT(T_2) has the required multiplicity information, this information cannot be used in an SQL query to "replicate" the tuples in V_7 the appropriate number of times. Thus, there is *no* rewriting of Q_7 that uses view V_7.[3] □

6 Related Work

There has been previous work on using views to answer queries (e.g., [YL87, SJGP90, TSI94, CR94, CKPS95, LMSS95]), but the problem of finding the equivalent rewritings for SQL queries with multiset semantics, grouping and aggregation, have received little attention.

Caching of previous query results was explored in [Sel88, SJGP90] as a means of supporting stored procedures. This corresponds to using materialized views when they match syntactically a sub-expression of the query. In the ADMS optimizer [CR94], subquery expressions corresponding to nodes in the query execution (operator) tree were also cached. A cached result was matched against a new query by using common expression analysis [Fin82]. Grouping and aggregation issues were not addressed.

View usability has been studied for conjunctive queries with set semantics and without grouping and aggregation in, e.g., [YL87, LMSS95]. Levy et al. [LMSS95] showed a close connection between the problem of usability of a view in evaluating a query and the problem of query containment. However, this connection does not carry over to the multiset case. [LMSS95] also presented a simple technique for generating a rewriting of a query Q using view V, under the set semantics. Essentially, the technique consists of first conjoining V to the FROM clause of Q, and then (independently) minimizing the resulting query to eliminate redundant tables. In the case of SQL queries, however, because of the multiset semantics, the query will not be equivalent after conjoining V to the FROM clause, even if it may be equivalent after removing other tables. Therefore, we need to find a priori which tables in the FROM clause will be replaced by V.

Optimization of conjunctive SQL queries using conjunctive views has been studied in [CKPS95]. In addition to considering when such views are usable in evaluating a query, they suggest how to perform this optimization in a cost-based fashion. However, they did not consider grouping and aggregation, nor did they consider the possibility of rewritings that are UNION ALLs of single-block queries.

[3] Gupta et al. [GHQ95] have suggested an "expand" operator to replicate tuples in a given table.

Recently, Gupta et al. [GHQ95] considered the problem of using materialized aggregation views to answer aggregation queries using a purely transformational approach. They perform syntactic transformations on the operator tree representation of the query such that the definition of the view would be identical to a sub-part of the definition of the query. Additional transformations on queries involving aggregation have been proposed by [YL94, LMS94, CS94, RSSS95, GHQ95, CS96, LM96]. The transformational approach is more restrictive than our semantic approach — in particular, the algorithm of Gupta et al. does not take the conditions in the WHERE and HAVING clauses into account when comparing $Sel(Q)$ with $Sel(V)$ and $Groups(Q)$ with $Groups(V)$ (see, e.g., conditions C_2^a and C_3^a). Further, their approach does not consider rewritings that are UNION ALLs of single-block queries. For example, their techniques would neither determine the usability of view V_1 in evaluating query Q_1 in Example 1.1, nor the usability of view V_1' in evaluating Q_1 in the same example. Also, Gupta et al. do not provide any formal guarantees of completeness.

A related problem is studied in Gupta et al. [GMR95]. They assume that a materialized view may be redefined, and investigate how to adapt the materialization of the view to reflect the redefinition. This problem is clearly a special case of the one we study, with the additional assumptions that the system knows the type of modification that took place, that the new view definition is "close" to the old definition, and that the view materialization may be modified.

7 Conclusions

The exploitation of materialized views is likely to be an important technique for performance enhancement, particularly for applications such as data warehousing where access to the base data is more expensive than access to the views. In this paper we presented general techniques to rewrite a given SQL query so that it uses materialized views, if possible.

We have focused on single-block SQL queries and views. Often, multi-block SQL queries that have view tables in the FROM clause can be transformed to single-block queries, e.g., using techniques described in [YL94, CS94, GHQ95, CS96]. In such cases, our techniques can also be applied.

We are currently extending our work in several ways, including considering the view usage problem for arbitrary nested queries, integrating our techniques with cost-based optimizers along the lines described in [CKPS95], and developing strategies for determining which views to cache.

References

[BI94] D. Barbará and T. Imieliński. Sleepers and workaholics: Caching strategies in mobile environments. In *Proc. ACM SIGMOD*, 1994.

[CKPS95] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proc. ICDE*, 1995.

[CR94] C. M. Chen and N. Roussopoulos. The implementation and performance evaluation of the ADMS query optimizer: Integrating query result caching and matching. In *Proc. EDBT*, 1994.

[CS94] S. Chaudhuri and K .Shim. Including group-by in query optimization. In *Proc. VLDB*, 1994.

[CS96] S. Chaudhuri and K .Shim. Optimizing queries with aggregate views. In *Proc. EDBT*, 1996.

[CV93] S. Chaudhuri and M. Y. Vardi. Optimization of real conjunctive queries. In *Proc. ACM PODS*, 1993.

[Fin82] S. Finkelstein. Common expression analysis in database applications. In *Proc. ACM SIGMOD*, 1982.

[GHQ95] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *Proc. VLDB*, 1995.

[GJM96] A. Gupta, H. V. Jagadish, and I. S. Mumick. Data warehousing using self-maintainable views. In *Proc. EDBT*, 1996.

[GMR95] A. Gupta, I. S. Mumick, and K. A. Ross. Adapting materialized views after redefinitions. In *Proc. ACM SIGMOD*, 1995.

[JMS95] H. V. Jagadish, I. S. Mumick, and A. Silberschatz. View maintenance issues for the Chronicle data model. In *Proc. ACM PODS*, 1995.

[LM96] A. Y. Levy and I. S. Mumick. Reasoning with aggregation constraints. In *Proc. EDBT*, 1996.

[LMS94] A. Y. Levy, I. S. Mumick, and Y. Sagiv. Query optimization by predicate move-around. In *Proc. VLDB*, 1994.

[LMSS95] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proc. ACM PODS*, 1995.

[LRO96] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. VLDB*, 1996.

[LSK95] A. Y. Levy, D. Srivastava, and T. Kirk. Data model and query evaluation in global information systems. *Journal of Intelligent Information Systems*, 5:121–143, 1995. Special Issue on Networked Information Discovery and Retrieval.

[RSSS95] K. A. Ross, D. Srivastava, P. Stuckey, and S. Sudarshan. Foundations of aggregation constraints. An early version appeared in Proc. of the 2nd Intl. Workshop on Principles and Practice of Constraint Programming, 1994, LNCS 874, 1995.

[Sel88] T. Sellis. Intelligent caching and indexing techniques for relational database systems. *Information Systems*, pages 175–185, 1988.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in database systems. In *Proc. ACM SIGMOD*, 1990.

[TSI94] O. G. Tsatalos, M. H. Solomon, and Y. E. Ioannidis. The GMAP: A versatile tool for physical data independence. In *Proc. VLDB*, 1994.

[YL87] H. Z. Yang and P.-A. Larson. Query transformation for PSJ-queries. In *Proc. VLDB*, 1987.

[YL94] W. P. Yan and P.-A. Larson. Performing group-by before join. In *Proc. ICDE*, 1994.

[ZGMHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proc. ACM SIGMOD*, 1995.