Answering Queries with Aggregation Using Views

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

Shaull 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 [B194], 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 than 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

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, TS194, 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_Plan(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:

$$Q_1: \text{SELECT Year, Plan_Name, SUM(Charge)} \text{ FROM Calls, Calling_Plan} \text{ WHERE Calls.Plan_Id = Calling_Plan.Plan_Id AND Year \geq 1990 AND Year \leq 1995 GROUPBY Year, Plan_Name HAVING SUM(Charge) > 1,000,000}$$

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:

$$V_1: \text{SELECT Plan_Id, Month, Year, SUM(Charge) FROM Calls GROUPBY Plan_Id, Month, Year}$$

View $V_1$ can be used to evaluate the query $Q_1$ by joining $V_1$ with the table Calling_Plan, collapsing multiple groups corresponding to the monthly plan earnings into annual plan earnings, and enforcing the additional conditions to get the summaries of plans that have earned more than a million dollars in one of the years between 1990 and 1995. The rewritten query $Q'_1$ that uses $V_1$ is:

$$Q'_1: \text{SELECT Year, Plan_Name, SUM(Earnings) FROM V_1, Calling_Plan WHERE V_1.Plan_Id = Calling_Plan.Plan_Id AND Year \geq 1990 AND Year \leq 1995 GROUPBY Year, Plan_Name HAVING SUM(Earnings) > 1,000,000}$$

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:

$$V'_1: \text{SELECT \ldots FROM \ldots GROUPBY \ldots}$$

The capacity of expressions that can be used to describe an aggregate view has been limited to union and projection. We show that it is possible to use the views to evaluate the query $Q_1$ by joining $V'_1$ with the table Calling_Plan, collapsing multiple groups corresponding to the monthly plan earnings into annual plan earnings, and enforcing the additional conditions to get the summaries of plans that have earned more than a million dollars in one of the years between 1990 and 1995. The rewritten query $Q''_1$ that uses $V'_1$ is:

$$Q''_1: \text{SELECT Year, Plan_Name, SUM(Earnings) FROM V'_1, Calling_Plan WHERE V'_1.Plan_Id = Calling_Plan.Plan_Id AND Year \geq 1990 AND Year \leq 1995 GROUPBY Year, Plan_Name HAVING SUM(Earnings) > 1,000,000}$$

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 $V'_1$.
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

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 $R.A$ in $Q$ is replaced by $A_1$, and every reference to $R.A$ in $Q$ is replaced by $A_2$.

We use $Tables(Q)$ to denote the set of tables (along with their columns) $\{R_1(\tilde{A}_1), \ldots, R_n(\tilde{A}_n)\}$ in the FROM clause of a single-block query $Q$, and $Cols(Q)$ to denote $\tilde{A}_1 \cup \ldots \cup \tilde{A}_n$, 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 $\{A, MAX(D), SUM(E)\}$, $ColSel(Q_e)$ 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 $\alpha \ op \ \beta$, where $\alpha$ and $\beta$ are terms formed from columns of tables, aggregation columns and constants using the arithmetic operations $+$ and $\ast$.

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 $SUM(D) > 1000$.

Given a single-block query $Q$, if $Groups(Q)$, $AggSel(Q)$ and $GConds(Q)$ are empty, 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 $\phi$ 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 $\phi$ 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

Note that each of $Groups(Q_e)$, $AggSel(Q_e)$ and $GConds(Q_e)$ can be empty without the other two being empty.
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 $Q'_3$ 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 $Q$.
- $V$ does not discard any tuples needed by $Q$.

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 $\phi$; 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 C1: There is a 1-1 column mapping \( \phi \) from \( V \) to \( Q \).

Condition C2: If a column \( A \) in \( \text{ColSel}(Q) \cup \text{Groups}(Q) \) is a column in \( \phi(\text{Cols}(V)) \), then \( \text{Sel}(V) \) must have a column \( B_A \) such that \( \text{Conds}(Q) \) implies \( (A = \phi(B_A)) \).

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

Condition C3: Suppose \( \text{AGG}(A) \) is in \( \text{Sel}(Q) \). If column \( A \) is in \( \phi(\text{Cols}(V)) \), then:

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

Condition C4: There exists a boolean combination of built-in predicates, \( \text{Conds}' \), such that:

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

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 C3 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 \( \phi(\text{Tables}(V)) \) by \( V(\phi(V)) \) in the FROM clause, where \( \phi(V) \) denotes \( V(\phi(\text{Sel}(V))) \). The SELECT and WHERE clauses of the query are then modified to reflect the use of \( V \) in the rewritten query. Formally, the single-block rewritten query \( Q' \) is obtained from \( Q \) by applying algorithm \textit{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 \textit{ConjViewSingleBlock}, is a rewriting of \( Q \) using \( V \).

If \( \text{Conds}(Q) \) and \( \text{Conds}(V) \) 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 \textit{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.

\[
Q_2: \text{SELECT } PN_l, \text{SUM}(C1), \text{COUNT}(C1) \text{ FROM } \text{Calls}(F_1, T_1, T_{Il}, D_1, M_1, Y_1, DU_1, P_1, C_1), \text{CallingPlans}(P_I, P_{NI}) \text{ WHERE } P_1 = P_{NI} \text{ AND } Y_1 = 1995 \text{ AND } M_1 = 12 \text{ GROUPBY } P_{NI}
\]

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

\[
V_2: \text{SELECT } F_2, T_2, T_{I2}, D_2, M_2, Y_2, DU_2, P_2, C_2 \text{ FROM } \text{Calls}(F_2, T_2, T_{I2}, D_2, M_2, Y_2, DU_2, A, C_2) \text{ WHERE } \& = \&_5 \text{ AND } M_2 = 12
\]

View \( V_2 \) can be used to evaluate query \( Q_2 \) since conditions \( C_1 - C_4 \) are satisfied: (C1) The 1-1 column mapping \( \phi \) from \( V_2 \) to \( Q_2 \) is \{ \( F_2 \rightarrow F_1, T_2 \rightarrow T_1, T_{I2} \rightarrow T_{Il}, D_2 \rightarrow D_1, M_2 \rightarrow M_1, Y_2 \rightarrow Y_1, DU_2 \rightarrow DU_1, P_2 \rightarrow P_1, C_2 \rightarrow C_1 \} \). (C2) Trivially satisfied. (C3) For column \( C_1, B_{C_1} \) is the column \( C_2 \) in \( \text{Sel}(V_2) \). (C4) \( \text{Conds}' \) is given by \( P_1 = P_{NI} \).

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

\[
Q'_2: \text{SELECT } PN_l, \text{SUM}(C1), \text{COUNT}(C1) \text{ FROM } V_2(F_1, T_1, T_{I1}, D_1, M_1, Y_1, DU_1, P_1, C_1), \text{CallingPlans}(P_I, P_{NI}) \text{ WHERE } P_1 = P_{NI} \text{ GROUPBY } P_{NI}
\]

Algorithm \textit{ConjViewSingleBlock}

Step S1: Replace all the tables in \( \phi(\text{Tables}(V)) \) by \( V(\phi(V)) \).
Step S2: Replace each column \( A \) in \( \text{Groups}(Q) \cup \text{ColSel}(Q) \cup \text{AggSel}(Q) \) by \( \phi(B_A) \), where \( B_A \) satisfies conditions \( C_1 \) and \( C_3 \), part 1.
Step S3: Determine a boolean combination of built-in predicates \( \text{Conds}' \) satisfying condition \( C_4 \). Replace \( \text{Conds}(Q) \) in \( Q \) by \( \text{Conds}' \).
Step S4: Consider an aggregation column COUNT(A) in \( \text{Sel}(Q) \) such that \( A \) is in \( \phi(\text{Cols}(V)) \), but not in \( \phi(\text{Sel}(V)) \). 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.

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 $C'^m$ : Let $\text{Conds}_1$ be $\text{Conds}(Q) \land \phi(\text{Conds}(V))$, and $\text{Conds}_2$ be $\text{Conds}(Q) \land \neg \phi(\text{Conds}(V))$. Then,

1. $\text{Conds}_1$ must be satisfiable.
2. There exists a boolean combination of built-in predicates, $\text{Conds}'$, such that:
   
   (a) $\text{Conds}_1$ is equivalent to $\phi(\text{Conds}(V)) \land \text{Conds}'$,
   
   (b) $\text{Conds}'$ involves only the columns in $\phi(\text{Sel}(V)) \cap \text{Cols}(Q) - \phi(\text{Conds}(V))$.
   
   (c) $\forall f_{\text{Group}}(\text{Conds}(Q))$ and $\forall f_{\text{Group}}(\text{Conds}(Q))$ 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, $C'^m$, 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 two single-block queries $Q_a$ and $Q_b$, that differ from $Q$ (and from each other) only in their WHERE clauses, such that:

1. $\text{Conds}(Q_a)$ is equivalent to $\phi(\text{Conds}(V))$ and $\text{Conds}_1$ is equivalent to $\text{Conds}(Q) \land \neg \phi(\text{Conds}(V))$.
2. $\text{Conds}_1$ involves only the columns in $\phi(\text{Sel}(V)) \cap \text{Cols}(Q) - \phi(\text{Conds}(V))$.
3. $\forall f_{\text{Group}}(\text{Conds}(Q))$ and $\forall f_{\text{Group}}(\text{Conds}(Q))$ is FALSE.

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

**Algorithm ConjViewMultiBlock**

**Step $S^m$** : Use $\phi(\text{Conds}(V))$ to split $Q$ into $Q_a$ and $Q_b$, such that $\text{Conds}(Q_a) = \phi(\text{Conds}(V))$ and $\text{Conds}(Q_b) = \phi(\text{Conds}(V))$. Let $Q'_a$ denote the resultant single-block query.

**Step $S^n$** : Use algorithm ConjViewSingleBlock to rewrite $Q_a$ to make use of view $V$. Let $Q'_b$ denote the resultant single-block query.

**Step $S^n$** : 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 $Q_b$.

3.2 Multiple Uses of Views

Often a query can make use of multiple views, or the same view multiple times. The rewriting algorithms ConjViewM-

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 $C'^m$ 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 $V$. □

Theorem 3.3 Let $Q$ be a single-block aggregation query without a HAVING clause, and let $V_0, \ldots, V_m$ 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 $V_0, \ldots, V_m$, then the result of rewriting $Q$ to incorporate views $V_0, \ldots, V_m$ would be the same regardless of the order in which the views are considered.
4. If $\text{Conds}(Q), \text{Conds}(V_0), \ldots, \text{Conds}(V_m)$ 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 $V_0, \ldots, V_m$ can be obtained by iteratively applying algorithm ConjViewSingleBlock.
Condition $C_3^b$ : Suppose $\text{AGG}(A)$ is in $\text{Sel}(Q)$ or in $\text{GConds}(Q)$. If column $A$ is in $\phi(\text{Cols}(V))$, then:

1. If $\text{AGG}$ is MIN, MAX or SUM, then $\text{Sel}(V)$ must have a column $B_A$ such that $\text{Conds}(Q)$ implies $A = \phi(B_A)$.
2. If $\text{AGG}$ is COUNT, then $\text{Sel}(V)$ 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 $\text{ConjViewSingleBlock}$ 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 [LMS93], 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 $\text{ConjViewSingleBlock}$ need to be extended to account for:

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

To accommodate such conditions we modify $C_3$ to also consider arguments that appear in $\text{GConds}(Q)$. The extended condition, $C_3^b$, is formally presented in Figure 6. If $Q$ and $V$ satisfy conditions $C_1$, $C_2$, $C_3^b$ and $C_4$, the single-block rewritten query $Q'$ is obtained using algorithm $\text{HavingConjViewSingleBlock}$, 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 $C_1$, $C_2$, $C_3^b$ and $C_4$ are satisfied, $V$ is usable in evaluating $Q$. In that case $Q'$, obtained by applying

Algorithm HavingConjViewSingleBlock

Assume that the query $Q$ has been pre-processed.

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

Step $S_2^b$ : Replace each column $A$ in $\text{GConds}(Q)$ by $\phi(B_A)$, where $A$ and $B_A$ satisfy conditions $C_1$ and $C_3^b$, part 1.

Step $S_3^b$ : Consider an aggregation column $\text{COUNT}(A)$ in $\text{Sel}(Q)$ or in $\text{GConds}(Q)$ such that $A$ is in $\phi(\text{Cols}(V))$, but not in $\phi(\text{Sel}(V))$. Replace $\text{COUNT}(A)$ by $\text{COUNT}(B)$, 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 $Q'$ using $V$. ⊓⊔

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 $\text{Conds}(Q)$ given the conditions in $\text{GConds}(Q)$, and removing redundant conditions in $\text{GConds}(Q)$. These techniques can be applied to rewrite the query $Q$, as a pre-processing step, yielding possibly modified conditions $\text{Conds}(Q)$ and $\text{GConds}(Q)$. The modified $\text{Conds}(Q)$ and $\text{GConds}(Q)$ are then used in checking conditions $C_2$, $C_3^b$ 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 $F_3$, $\text{MAX}(C_1)$ FROM . Calls$(F_3, T_1, T_1, D_1, M_1, Y_1, DU_1, P_1, C_1)$, Calling_Plans$(P_1, P N_1)$ WHERE $P_1 = P_1$ AND $P N_1 = \text{"TrueUniverse"}$ AND $Y_1 = 1995$ AND $M_1 = 12$ GROUPBY $F_3$ HAVING $\text{MAX}(C_1) > 10$

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

\[
Q_4: \text{SELECT } P_1, PN_1, SUM(C_1), \text{MAX}(C_1) \\
\text{FROM } \text{Calls}(F_2, T_2, T_1, D_2, M_2, Y_2, DU_2, P_2, C_2), \\
\text{Calling-Plans}(PI_1, PN_1) \\
\text{WHERE } P_1 = PI_1 \text{ AND } Y_1 = 1995 \\
\text{GROUP BY } P_1, PN_1
\]

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:

\[
V_4: \text{SELECT } P_2, M_2, Y_2, SUM(ME_2), \text{MAX}(MC_2) \\
\text{FROM } \text{Calls}(F_2, T_2, T_1, D_2, M_2, Y_2, DU_2, P_2, C_2), \\
\text{Calling-Plans}(PI_1, PN_1) \\
\text{WHERE } P_1 = PI_1 \text{ AND } Y_1 = 1995 \\
\text{GROUP BY } P_2, M_2, Y_2
\]

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

\[
Q'_4: \text{SELECT } P_1, PN_1, SUM(ME_1), \text{MAX}(MC_1) \\
\text{FROM } V_4(P_1, M_1, Y_1, ME_1, MC_1), \\
\text{Calling-Plans}(PI_1, PN_1) \\
\text{WHERE } P_1 = PI_1 \text{ AND } Y_1 = 1995 \\
\text{GROUP BY } P_1, PN_1
\]

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:

\[
Q_5: \text{SELECT } P_1, \text{COUNT}(CN_1) \\
\text{FROM } \text{Calls}(F_1, T_1, T_1, D_1, M_1, Y_1, DU_1, P_1, C_1), \\
\text{Calling-Plans}(PI_1, PN_1) \\
\text{WHERE } F_1 = PN_1 \text{ AND } Y_1 = 1995 \\
\text{GROUP BY } P_1
\]

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

\[
V_{5a}: \text{SELECT } F_3, P_3, Y_3, SUM(C_3) \\
\text{FROM } \text{Calls}(F_3, T_3, T_1, D_3, M_3, Y_3, DU_2, P_3, C_3) \\
\text{GROUP BY } F_3, P_3, Y_3
\]
Although $V_{3a}$ does not project out any column that is needed in $Q_4$, $V_{3a}$ cannot be used to evaluate $Q_4$. This is because the multiplicity of the $\text{From}$ column of $Calls$ is needed in order to compute $\text{COUNT}(CN_1)$, but that multiplicity is lost in the view $V_{3a}$. However, consider view $V_{3b}$ below:

$$V_{3b}: \text{SELECT } F_2, P_2, Y_2, \text{SUM}(C_2), \text{COUNT}(C_2)$$
$$\text{FROM } Calls(F_2, T_2, TI_2, D_2, M_2, Y_2, DU_2, P_2, C_2)$$
$$\text{GROUP BY } F_2, P_2, Y_2$$

Although the multiplicities of the $\text{From}$ column are not explicit in $V_{3b}$, they can be computed using the available information. $V_{3b}$ can be used to evaluate $Q_3$ as follows:

$$Q_3: \text{SELECT } P_1, \text{SUM}(C_1)$$
$$\text{FROM } V_{3b}(F_1, P_1, Y_1, E_1, C_1),$$
$$\text{Customer}(P_N, C_1)$$
$$\text{WHERE } F_1 = P_N \text{ AND } Y_1 = 1995$$
$$\text{GROUP BY } P_1$$

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 $\text{ColSel}(Q)$ must be a subset of $\text{Groups}(Q)$, condition $C_2'$ is a generalization of condition $C_2$. Intuitively, condition $C_2'$ guarantees that the columns in the view contain enough information to compute the aggregates required in the query. In particular, conditions $C_2'$, 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 $\text{SUM}$ or $\text{COUNT}$). 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 $C_2'$ does not allow $\text{Conds}'$ to constrain any of the columns in $\phi(\text{AggSel}(V))$. Intuitively, this is because the columns in $\text{AggSel}(V)$ are aggregated upon in view $V$, and hence are not "available" for imposition of additional constraints in the rewritten query $Q'$.

If conditions $C_2' - C_4'$ are satisfied, the rewritten query $Q'$ is obtained from $Q$ by applying algorithm $\text{AggViewSingleBlock}$, presented in Figure 9. Steps $S_1'$, $S_2'$ and $S_3'$ are similar to steps $S_1$, $S_2$ and $S_3$ of algorithm $\text{ConjViewSingleBlock}$. Steps $S_4'$ and $S_5'$ 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 $C_1' - C_4'$ are satisfied, $V$ is usable in evaluating $Q$. In that case $Q'$, obtained by applying algorithm $\text{AggViewSingleBlock}$, 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 $C_1' - C_4'$ are satisfied.

**Condition $C_1'$:** The 1-1 column mapping $\phi$ from $V_4$ to $Q_4$ is \{\$F_2 \to F_1, T_2 \to T_1, T_2 \to T_1, D_2 \to D_1, M_2 \to M_1, Y_2 \to Y_1, DU_2 \to DU_1, P_2 \to P_1, C_2 \to C_1\}.

**Condition $C_2'$:** For column $P_1$ in $\text{Groups}(Q_4)$, $B_{P_1}$ is the column $P_2$ in $\text{ColSel}(V_4)$.

**Condition $C_3'$:** For column $\text{SUM}(C_1)$ in $\text{Sel}(Q_4)$, $\text{Sel}(V_4)$ contains column $\text{SUM}(C_2)$, and for column $\text{MAX}(C_1)$ in $\text{Sel}(Q_4)$, $\text{Sel}(V_4)$ contains column $\text{MAX}(C_2)$.

**Condition $C_4'$:** $\text{Conds}'$ is the same as $\text{Conds}(Q_4)$, i.e., $P_1 = P_{II} \text{ AND } Y_1 = 1995$ since no conditions are enforced in $V_4$.

The rewritten query $Q'$ resulting from applying steps $S_1'$ - $S_5'$ is given in Example 4.1.

**Example 4.4 (Constraining $\phi(\text{AggSel}(V))$)**

Consider again the telephone company database from Example 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 $1$.

$$Q_6: \text{SELECT } A, \text{SUM}(C_1)$$
$$\text{FROM } Calls(F_1, T_1, T_1, D_1, M_1, Y_1, DU_1, A, C_1)$$
$$\text{WHERE } Y_1 = 1995 \text{ AND } C_1 > 1$$
$$\text{GROUP BY } P_1$$

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

$$V_6: \text{SELECT } P_2, M_2, Y_2, \text{SUM}(C_2), \text{MAX}(C_2)$$
$$\text{FROM } Calls(F_2, T_2, T_2, D_2, M_2, Y_2, DU_2, P_2, C_2)$$
$$\text{GROUP BY } P_2, M_2, Y_2$$

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 $V_4$; no condition on the result of the $\text{SUM}$ or the $\text{MAX}$ in $V_6$ can capture the effect of the condition on $C_1$ in $Q_6$. □
Condition $C^+_t$ : Same as condition $C_t$.

Condition $C^+_2$ : 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 $C^+_3$ : Suppose $AGG(A)$ is in $Sel(Q)$.

1. If column $A$ is in $\phi(\phi(Cols(V)))$, 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 $\phi(\phi(Cols(V)))$, 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 $C^+_4$ : There exists $Conds'$, such that:

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

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 $S^+_1$ : 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 $S^+_2$ : 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(a).

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

Step $S^+_3$ : Determine a boolean combination of built-in predicates $Conds'$ satisfying condition $C^+_4$ as above. Replace $Conds(Q)$ in $Q$ by $Conds'$.

Step $S^+_4$ : Consider an aggregation column $AGG(A)$ in $Sel(Q)$ such that $A$ is in $\phi(\phi(Cols(V)))$.

1. Let $AGG$ be MIN, MAX or SUM. By condition $C^+_4$, 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 $\phi(S)$.
   (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 $C^+_3$, 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 $C^+_4$, 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 $S^+_5$ : Consider an aggregation column $AGG(A)$ in $Sel(Q)$ such that column $A$ is not in $\phi(\phi(Cols(V)))$.

If $AGG$ is MIN or MAX, leave $AGG(A)$ unchanged.
If $AGG$ is SUM or COUNT, do the following: By condition $C^+_4$, 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

327
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 $C^*_2$ 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 $Group(V)$ 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 $S^*_2$ determines a $GConds'$ in addition to $Conds'$, using $GConds'(Q)$ and $GConds'(V)$ (resulting from the pre-processing step). Steps $S^*_2$ and $S^*_1$ 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:

$$Q_7: \begin{align*}
\text{SELECT} & \quad F_1, D_1, M_1, Y_1 \\
\text{FROM} & \quad \text{Calls}(F_1, T_1, T_1, D_1, M_1, Y_1, DU_1, P_1, C_1) \\
\text{WHERE} & \quad DU_1 > 3600
\end{align*}$$

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

$$V_7: \begin{align*}
\text{SELECT} & \quad F_2, D_2, M_2, Y_2, \text{COUNT}(T_2) \\
\text{FROM} & \quad \text{Calls}(F_2, T_2, T_1, D_2, M_2, Y_2, DU_2, P_2, C_2) \\
\text{WHERE} & \quad DU_2 > 3600 \\
\text{GROUPBY} & \quad F_2, D_2, M_2, Y_2
\end{align*}$$

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 $\text{COUNT}(T_2)$ 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 $V_7$.

6 Related Work

There has been previous work on using views to answer queries (e.g., [YL87, SI97, TS94, 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, SI97] 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 in 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 by 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 $C_2^* \text{ and } C_3^*$). Further, their approach does not consider rewrites that are UNION ALLs of single-block queries. For example, their techniques would not 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


