A Rewriting Algorithm for Multi-Block Aggregation Queries and Views using Prerequisites and Compensations

David DeHaan*
School of Computer Science, University of Waterloo
dedehaan@uwaterloo.ca

Technical Report CS-2004-25
May 3, 2004

*This work partially supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada, and by the University of Waterloo. The author would also like to thank his advisor Frank Wm. Tompa for many helpful discussions related to this work.

Abstract

This paper presents an extension to previously proposed bottom-up compensation-based algorithms for calculating the usability of a view to answer a database query, where both query and view definition are multi-block SQL queries containing aggregation. The goal of our extension is to increase the recall of previous algorithms. The main idea of approach is to improve recall by deferring certain view pruning decisions through the introduction of pseudo-algebraic operators that we call prerequisites. We present a set of transformation rules for manipulating query expressions containing prerequisite operators, as well as a modified matching algorithm that utilizes the prerequisite operators and associated transforms. Finally, we discuss the effect that prerequisite operators have on efficiency of the algorithm, and propose a combination of heuristics and normal forms to limit the introduction of prerequisites to scenarios in which they are likely to be helpful.

1 Introduction

View matching is the problem of determining whether or not a view V is useful for answering a query Q, and if so, how Q can be rewritten to use V. A view V is useful if its definition is contained as a subexpression within a query expression that is equivalent to Q. This paper details an algorithm for rewriting multi-block aggregation queries to use materialized views that are also defined by multi-block aggregation queries.

View matching is closely related to query equivalence. The query equivalence problem has been studied in the context of aggregation queries by Nutt et al.[15, 8, 6, 7]. However, these authors only consider queries with a single aggregation operator at the root of the query; as well, they do not consider any schema information.

View matching is also closely related to logical query rewriting. Many authors have suggested rules for transforming multi-block aggregation queries[16, 17, 3, 4, 2]; however, this work is orthogonal to our own. In fact, our algorithm for view matching uses a set of these transform rules as an extensible toolkit.

There are several papers in the literature which discuss view matching for SQL queries. Bello et al.[1] describe the implementation of view matching in Oracle, but their paper gives few details, and their matching method only appears to support single-block aggregation queries. Goldstein and Larson[11] address the more limited problem of deciding whether or not a view *exactly matches* a query. (An “exact match” occurs when the definitions of Q and V are identical modulo some very simple operations such as application of a predicate.) Finding exact matches is a simpler problem than general view matching; hence, the main contribution of their paper is an efficient indexing technique for speeding up exact matching. The exact matching test is invoked repeatedly by an external process as it navigates through the query definition. A weakness of this method is that the exact matching algorithm provides no guidance as to how the query definition should be manipulated using transformation rules in order to find a successful match. Thus, in order to answer view usability in a complete fashion, the algorithm of Goldstein and Larson must assume that an external transformational optimizer is driving the matching via a forward-chaining application of transformation rules that exhaustively generates all possible formulations of the query.

Zaharioudakis et al.[18] describe a bottom-up algorithm for view matching. In contrast to the work described above, the algorithm described by these authors handles multi-block view definitions. Their algorithm is presented as a set of simple templates that dictate the form of the rewrite, assuming that a matching template can be found. The authors do not explicitly utilize transformation rules, although each template essentially embodies the application of a sequence of transformation rules. A weakness of this method is that although it does not incur the expense of applying transformation rules in a forward-chaining fashion, it also is unable to find rewritings that could potentially be found by a strategy that does. Our work in this paper attempts to increase the matching power of the bottom-up algorithm of Zaharioudakis et al. by using transformation rules, but we use the matching process to guide the application of the transformation rules so that they are only ever applied in a goal-oriented manner.

The outline of this paper is as follows. Section 2 defines some preliminary concepts such as the query language and classes of aggregation functions considered. Section 3 gives an (incomplete) listing of known transformation rules that can be used to rewrite aggregation queries. Section 4 describes a generalized version of Zaharioudakis' algorithm for bottom-up view matching. Section 5 introduces a new concept that we call a *prerequisite* query operator, and Section 6 details a list of transformation rules for modifying queries containing prerequisite operators. Finally, Section 7 describes how prerequisite operators can be integrated into the bottom-up matching process in order to increase the matching power of the algorithm.

2 Preliminaries

2.1 Encoding Semantic Information

Suppose that a database catalog contains semantic information encoded as integrity constraints. We will consider two common types of constraints: functional dependencies and inclusion dependencies. Let Σ be a set of functional dependencies of the form

$$Y \rightarrow z$$

where Y is a set of attributes of some relation R, and z is a single attribute of R[1]. To accommodate bag semantics, we will assume that each (base or intermediate) relation R has a virtual attribute Id_R. Then, for each attribute z in base relation R, Σ contains the functional dependency $\{Id_R\} \rightarrow z$; as well, for each set of attributes Y forming a candidate key of R, Σ contains the functional dependency $Y \rightarrow Id_R$.

Let Δ be a set of inclusion dependencies of the form

$$R_1(x_1) \subseteq R_2(x_2)$$

where R_1 and R_2 are relation names, x_1 is an attribute of R_1, and x_2 is an attribute of R_2. Then, the implication problem with respect to $\Sigma \cup \Delta$ is in NP[13] (note that if Δ is not restricted to unary inclusion dependencies, the implication problem becomes undecidable[14]).

2.2 Query Language

The query language that we consider for both query and view definitions is a subset of conjunctive SQL including aggregation. Informally, define a *single-block* SQL query as an SQL query composed of selections, projections, and joins of (base) relations, followed by an optional aggregation operation (GROUP BY + aggregation functions + HAVING clause); a *multi-block* query is then inductively defined as single block query in which the base relations are generalized to being either single or multi-block queries. In this paper we do not consider NULL values, and so all joins are presumed to be inner joins.

For conciseness, we will represent SQL queries using an algebraic notation that contains only two operators. The operator Π encapsulates selection, projection, and join operations (often called SPJ or PSJ queries); it is defined according to the following expansion into SQL

$$\Pi^{\alpha}_{\rho}(X_1, \ldots, X_n) \quad \equiv \quad \begin{array}{l} \texttt{SELECT } \alpha \\ \texttt{FROM } X_1, \ldots, X_n \\ \texttt{WHERE } \rho \end{array}$$

where

[1]Throughout this paper we may abuse notation slightly by allowing z to be a set of attributes, which is to be interpreted as a set of FDs, one for each member of z.

- each X_i is either a base relation or subquery whose attribute set is described by the function $schema(X_i)$[2];
- α is a set of attribute names occurring in $\bigcup_i schema(X_i)$; and
- ρ is a set of binary predicates of the form $y\,\theta\,z$ where $y \in \bigcup_i schema(X_i)$ and either $z \in \bigcup_i schema(X_i)$ or $z \in D$ for some fixed domain of constants D.

Any predicate $y\,\theta\,z$ in ρ such that $y \in schema(X_i), z \in schema(X_j), i \neq j$ is referred to as a *join predicate* between X_i and X_j. Note that the function $schema(\cdot)$ is defined on Π as

$$schema(\textstyle\prod_{\rho}^{\alpha}(X_1, \ldots, X_n)) := \alpha$$

The operator Λ performs a grouping operation as well as calculating a set of aggregation functions over the tuples in each group. It is defined by the following expansion into SQL

$$\Lambda_F^{\alpha}(X_1) \quad \equiv \quad \begin{array}{l}\texttt{SELECT } \alpha\texttt{, } F \\ \texttt{FROM } X_1 \\ \texttt{GROUP BY } \alpha\end{array}$$

where α and X_1 are defined as for Π, and F is a set of aggregation expressions of the form "$f(x)$ as y" where $f(\cdot)$ is an aggregation function, x (the *internal expression*) is an expression composed of constants from domain D and attribute names from $schema(X)$, and y (the *external name*) is a valid attribute name that is unique from all other attribute names in the query. The function $schema(\cdot)$ is defined on Λ as

$$schema(\Lambda_F^{\alpha}(X_1)) := \alpha \cup external(F)$$

where $external(F)$ is a function returning the set of external names in F (similarly, $expressions(F)$ returns the set of internal expressions in F, and $attributes(F)$ returns the set of attribute names from $schema(X_1)$ occurring somewhere within $expressions(F)$).

With certain restrictions[3], the language formed by composing Π and Λ operators is equivalent to the multi-block SQL queries described earlier in this section. Note that HAVING clauses and grouping attributes which do not occur in the output attribute set can be modeled using a Π operator above a Λ operator.

Example 1 Suppose that you have a student table `S(sid, sname)` and an enrollment table `E(student, course, term, grade, credits)`. The SQL query

```
Q :=  SELECT sname, AVG(grade) as avg
      FROM S, E
      WHERE sid = student
      GROUP BY sid, sname
      HAVING SUM(credits) > 10
```

can be expressed by the algebraic expression

$$\mathtt{Q} := {}^{\mathtt{Q}}\textstyle\prod_{\{\mathtt{sum>10}\}}^{\{\mathtt{sname,avg}\}}({}^{\mathtt{Q_1}}\Lambda_{\{\mathtt{AVG(grade)\ as\ avg,SUM(credits)\ as\ sum}\}}^{\{\mathtt{sid,sname}\}}({}^{\mathtt{Q_{1.1}}}\textstyle\prod_{\{\mathtt{sid=student}\}}^{\{\mathtt{sid,sname,grade,credits}\}}(\mathtt{S},\mathtt{E})))$$

Note that each algebraic operator might be prefixed with a label in order to facilitate discussion.

[2]For simplicity, we assume that the attribute sets of all X_i are disjoint, which can be easily implemented by prefixing as needed the names of attributes with the relation or subquery from which they are drawn.

[3]Specifically, for simplicity of exposition the operators Π and Λ do not provide capabilities for renaming attributes or for outputting expressions; however, this functionality could be easily added.

2.3 Classes of Aggregation Functions

This section defines some useful classes of aggregation functions.

Definition 2.1 (Duplicate Insensitive Aggregation Function)
An aggregation function f is duplicate insensitive *if and only if its application over a bag of constants is equivalent to its application over the set projection of the same bag of constants. In other words, for any bag of constants B,*

$$f(B) = f(\{v : v \in B\}).$$

Common examples of duplicate insensitive functions include `Min`, `Max`, and `CountDistinct`.

Definition 2.2 (Cleanly Composable Functions) *[9]*
A pair of aggregation functions f and g cleanly compose *(written $f \circ g$) if and only if for any two bags of constants B_1 and B_2,*

$$f(\{\!|g(B_1 \uplus B_2)|\!\}) = f(\{\!|g(B_1), g(B_2)|\!\}).$$

where $\{\!|x, y|\!\}$ denotes a bag containing the elements x and y.

Some examples of cleanly composable aggregation functions include `Sum`∘`Count`, `Sum`∘`Sum`, and `Min`∘`Min`.

Definition 2.3 (Distributive Aggregation Function)
An aggregation function f is distributive *if there exists a pair of cleanly composable functions f_2 and f_1 such that $(f_2 \circ f_1) \equiv f$.*

For example, the aggregation function `Sum` is distributive because it is equivalent to the cleanly composable pairing `Sum`∘`Sum`, while `Count` is distributive because it is equivalent to `Sum`∘`Count`.

3 Transformation Rules

The rules in this section embody transformations that can be applied to transform a valid algebraic expression to another form that is also a valid algebraic expression and is equivalent to the first form with respect to interpretation under bag semantics. Since query equivalence is maintained, each rule could potentially be applied in both directions; however, most of the rules are presented as asymmetric both because in practice they are primarily useful in one direction, and because in many cases the the conditions that need to be tested to verify correctness are expressed with respect to the query expression on one side of the equivalence and would need to be re-formulated in order to use the rule in the opposite direction.

Let Σ be a set of functional dependencies and Δ be a set of unary inclusion dependencies such that $\Sigma \cup \Delta$ encode the integrity constraints which all valid instances of the database must satisfy. When the conditions of a transform rule refer to $\Sigma \cup \Delta$, query equivalence is only preserved with respect to interpretation over database instances that satisfy $\Sigma \cup \Delta$. For the remainder of this section, Q denotes a subquery expression, R a base relation, X either a subquery or a base relation, and $\overline{\mathbf{X}}$ a set of X.

3.1 Query Simplifying Transformations

3.1.1 View Flattening

View flattening is used to merge adjacent operators of the same type into a single block[4], or to partition a block into adjacent blocks.

[4]Since our defined algebraic operators correspond closely to blocks within an equivalent graphical representation of a query such as QGM[12], we refer to an instance of Π or Λ as a block, and to the entire query as a query graph.

Rule `VF1`

$$\begin{gathered}\Pi^{\alpha}_{\rho}(\overline{\mathbf{X}}_1, \Pi^{\beta}_{\varrho}(\overline{\mathbf{X}}_2)) \\ \Updownarrow \\ \Pi^{\alpha}_{\rho \cup \varrho}(\overline{\mathbf{X}}_1, \overline{\mathbf{X}}_2)\end{gathered}$$

Conditions: In the upward direction, the choice of β and ϱ must be syntactically valid.

Rule `VF2`

$$\begin{gathered}\Lambda^{\alpha}_{F^2}(\Lambda^{\beta}_{F^1}(X)) \\ \Updownarrow \\ \Lambda^{\alpha}_{F^{2\circ 1}}(X)\end{gathered}$$

Conditions:

1. $\alpha \subseteq \beta$
2. All of the aggregation expressions in $F^{2\circ 1}$ are decomposable and equivalent to aggregation expressions F^2 composed with aggregation expressions F^1.

3.1.2 Output Schema Pruning

These rules remove unnecessary attributes from the output schema of a subquery. Note that they can be trivially modified for the case where the parent operator is Λ instead of Π.

Rule `AR1`

$$\begin{gathered}\Pi^{\alpha}_{\rho}(\overline{\mathbf{X}}_1, \Pi^{\beta}_{\varrho}(\overline{\mathbf{X}}_2)) \\ \Downarrow \\ \Pi^{\alpha}_{\rho}(\overline{\mathbf{X}}_1, \Pi^{\beta'}_{\varrho}(\overline{\mathbf{X}}_2))\end{gathered}$$

where β' is defined as the subset of attribute names in β that occur either in α or in ρ. (In other words, $\beta' := \beta \cap (\alpha \cup \mathit{attributes}(\rho))$, where $\mathit{attributes}(\rho)$ is the set of attribute names occurring in the predicates in ρ.)

Conditions: none.

Rule `AR2`

$$\begin{gathered}\Pi^{\alpha}_{\rho}(\overline{\mathbf{X}}_1, \Lambda^{\beta}_{F}(\overline{\mathbf{X}}_2)) \\ \Downarrow \\ \Pi^{\alpha}_{\rho}(\overline{\mathbf{X}}_1, \Lambda^{\beta}_{F'}(\overline{\mathbf{X}}_2))\end{gathered}$$

where $F' := \{f \in F \mid \mathit{external}(f) \subseteq (\alpha \cup \mathit{attributes}(\rho))\}$.

Conditions: none.

Rule `AR3`

$$\begin{gathered}\Pi^{\alpha}_{\rho}(\overline{\mathbf{X}}_1, \Lambda^{\beta}_{F}(\overline{\mathbf{X}}_2)) \\ \Downarrow \\ \Pi^{\alpha}_{\rho}(\overline{\mathbf{X}}_1, \Lambda^{\beta'}_{F}(\overline{\mathbf{X}}_2))\end{gathered}$$

where $\beta' := \beta \cap (\alpha \cup \mathit{attributes}(\rho))$.

Conditions:

1. For each $x \in (\beta \setminus \beta')$, $\Sigma \cup \Delta$ plus any predicates in $\overline{\mathbf{X}}_2$ imply that $\beta' \rightarrow x$.

3.1.3 Conjunct Removal

This rule uses schema information to remove an "unnecessary" conjunct from an Π operator (that is, a conjunct that does not contribute any attributes to the output set, and is joined with a *lossless join*).

Rule `CR`

$$\Pi^{\alpha}_{\rho}(\overline{\mathbf{X}}, X_1)$$
$$\Downarrow$$
$$\Pi^{\alpha}_{\rho \setminus \rho'}(\overline{\mathbf{X}})$$

where ρ' is the subset of ρ that reference attributes in $\mathit{schema}(X_1)$.

Conditions:

1. $\alpha \cap \mathit{schema}(X_1) = \emptyset$
   That is, X_1 does not contribute to the output attribute set.
2. $\overline{\mathbf{X}} \bowtie X_1$ is a lossless join. More formally, let A and B be the sets of attribute names from X_1 and $\overline{\mathbf{X}}$, respectively, appearing in ρ'. Then,
   (a) All of the predicates in ρ' are equality join predicates between X_1 and $\overline{\mathbf{X}}$ (i.e. no filtering predicates over only X_1); and
   (b) $\Sigma \cup \Delta$ plus any predicates in $\overline{\mathbf{X}}$ and X_1 imply that
      i. $A \rightarrow \mathit{Id}_{X_1}$ and
      ii. $\overline{\mathbf{X}}(B) \subseteq X_1(A)$.

3.2 Group-by Pushdown

3.2.1 Invariant Grouping

Rule `PD1` [16]

$$\Lambda^{\alpha}_{F}(\Pi^{\beta}_{\rho}(\overline{\mathbf{X}}, X_1))$$
$$\Downarrow$$
$$\Pi^{\alpha \cup \mathit{external}(F)}_{\rho'}(X_1, \Lambda^{\alpha'}_{F}(\Pi^{\beta'}_{\rho \setminus \rho'}(\overline{\mathbf{X}})))$$

where ρ' is defined as the set of predicates in ρ that reference attributes in $\mathit{schema}(X_1)$. Let A be the set of attributes from $\overline{\mathbf{X}}$ that participate in the join to X_1 (that is, $A := \mathit{schema}(\overline{\mathbf{X}}) \cap \mathit{attributes}(\rho')$). Then, $\beta' := (\beta \cap \mathit{schema}(\overline{\mathbf{X}})) \cup A$, and $\alpha' := (\alpha \cap \mathit{schema}(\overline{\mathbf{X}})) \cup A$.

Conditions:

1. $\Sigma \cup \Delta$ plus the predicates in ρ and inside $\overline{\mathbf{X}}$ imply that $\alpha \rightarrow A$.
2. $\Sigma \cup \Delta$ plus the predicates in ρ and inside $\overline{\mathbf{X}}$ and X_1 imply that $\alpha \cup A \rightarrow \mathit{Id}_{X_1}$.

3.2.2 Eager Grouping

Rule `PD2` [17]
Assume that in the following query, $F_{\overline{\mathbf{X}}}$ and F_{X_1} are sets of aggregation expressions whose internal expression only reference constants as well as attributes from $\overline{\mathbf{X}}$ and X_1, respectively.

$$\Lambda^{\alpha}_{F_{\overline{\mathbf{X}}} \cup F^{2 \circ 1}_{X_1}}(\Pi^{\beta}_{\rho}(\overline{\mathbf{X}}, X_1))$$
$$\Downarrow$$
$$\Lambda^{\alpha}_{F_{\overline{\mathbf{X}}} \cup F^{2}_{X_1}}(\Pi^{\alpha \cup \mathit{attributes}(F_{\overline{\mathbf{X}}} \cup F^{2}_{X_1})}_{\rho \setminus \rho'}(\overline{\mathbf{X}}, \Lambda^{\alpha'}_{F^{1}_{X_1}}(\Pi^{\beta'}_{\rho'}(X_1))))$$

where ρ' is defined as any set of predicates in ρ that reference only attributes in $\mathit{schema}(X_1)$. Let A be the set of attributes from X_1 needed for the predicates pushed above the inner aggregation (that is, $A := \mathit{schema}(X_1) \cap \mathit{attributes}(\rho \setminus \rho')$). Then, $\beta' := (\mathit{schema}(X_1) \cap \alpha) \cup A \cup \mathit{attributes}(F^1_{X_1})$ and $\alpha' := (\mathit{schema}(X_1) \cap \alpha) \cup A$. Note that the innermost Π operator is unnecessary if ρ' is empty.

Conditions:

1. All of the aggregation expressions in $F^{2\circ 1}_{X_1}$ are decomposable and equivalent to aggregation expressions $F^2_{X_1}$ composed with aggregation expressions $F^1_{X_1}$.
2. All of the aggregation expressions in $F_{\overline{\mathbf{X}}}$ are duplicate insensitive aggregation functions.

3.2.3 Partial Eager Grouping

Rule `PD3` [9]
Assume that in the following query, $F_{\overline{\mathbf{X}}_2}$ and $F^1_{\overline{\mathbf{X}}_3}$ are sets of aggregation expressions whose internal expressions only reference constants as well as attributes from $\overline{\mathbf{X}}_2$ and $\overline{\mathbf{X}}_3$, respectively, and $F^2_{\overline{\mathbf{X}}_3}$ is a set of aggregation expression whose internal expressions only reference constants as well as attributes from $\mathit{external}(F^1_{\overline{\mathbf{X}}_3})$.

$$\Lambda^{\alpha}_{F_{\overline{\mathbf{X}}_2} \cup F^2_{\overline{\mathbf{X}}_3}}(\Pi^{\beta}_{\rho}(\overline{\mathbf{X}}_2, \Lambda^{\gamma}_{F^1_{\overline{\mathbf{X}}_3}}(\Pi^{\lambda}_{\varrho}(\overline{\mathbf{X}}_3, X_1))))$$
$$\Downarrow$$
$$\Lambda^{\alpha}_{F_{\overline{\mathbf{X}}_2} \cup F^2_{\overline{\mathbf{X}}_3}}(\Pi^{\beta}_{\rho \cup (\varrho \setminus \varrho')}(X_1, \overline{\mathbf{X}}_2, \Lambda^{\gamma'}_{F^1_{\overline{\mathbf{X}}_3}}(\Pi^{\lambda'}_{\varrho'}(\overline{\mathbf{X}}_3))))$$

where ϱ' is defined as any set of predicates in ϱ that reference only attributes in $\mathit{schema}(\overline{\mathbf{X}}_3)$. Let A be the set of attributes from $\overline{\mathbf{X}}_3$ needed for the predicates pushed above the inner aggregation (that is, $A := \mathit{schema}(\overline{\mathbf{X}}_3) \cap \mathit{attributes}(\varrho \setminus \varrho')$). Then, $\gamma' := (\gamma \cap \mathit{schema}(\overline{\mathbf{X}}_3)) \cup A$ and $\lambda' := (\lambda \cap \mathit{schema}(\overline{\mathbf{X}}_3)) \cup A$.

Conditions:

1. All of the expressions in $F^2_{\overline{\mathbf{X}}_3}$ cleanly compose with the expressions in $F^1_{\overline{\mathbf{X}}_3}$ which they depend upon.
2. All of the aggregation expressions in $F_{\overline{\mathbf{X}}_2}$ are duplicate insensitive aggregation functions.

3.3 Group-by Pullup

3.3.1 Invariant Grouping

Rule `PU1` [4] (essentially an adaptation of [16])

$$\Pi^{\beta}_{\rho}(X_1, \overline{\mathbf{X}}_2, \Lambda^{\gamma}_{F}(X_3))$$
$$\Downarrow$$
$$\Pi^{\beta}_{\rho \setminus \rho'}(\overline{\mathbf{X}}_2, \Lambda^{\gamma \cup \delta}_{F}(\Pi^{\lambda \cup \delta}_{\rho'}(X_1, X_3)))$$

where ρ' is defined as the set of predicates in ρ that reference only attributes in $\mathit{schema}(X_1) \cup \mathit{schema}(X_3)$, and δ is any set of attribute names from $\mathit{schema}(X_1)$ that contains all of the necessary join and output attributes of X_1—that is, $\delta \supseteq (\beta \cup \mathit{attributes}(\rho \setminus \rho')) \cap \mathit{schema}(X_1)$.

Conditions:

1. ρ does not contain any predicates that reference external names from F.
2. $\Sigma \cup \Delta$ plus predicates in ρ, X_1, $\overline{\mathbf{X}}_2$, and X_3 imply that $\gamma \cup \delta \rightarrow \mathit{Id}_{X_1}$.

3.3.2 Lazy Grouping

Rule `PU2` (adapted from [17])

Assume that in the following query, $F_{\overline{\mathbf{X}}}$ and F_{X_1} are sets of aggregation expressions whose internal expression only reference constants as well as attributes from $\overline{\mathbf{X}}$ and X_1, respectively.

$$\Lambda^{\alpha}_{F_{\overline{\mathbf{X}}} \cup F^2_{X_1}}(\Pi^{\beta}_{\rho}(\overline{\mathbf{X}}, \Lambda^{\gamma}_{F^1_{X_1}}(X_1)))$$
$$\Downarrow$$
$$\Lambda^{\alpha}_{F_{\overline{\mathbf{X}}} \cup F^{2\circ 1}_{X_1}}(\Pi^{\beta}_{\rho}(\overline{\mathbf{X}}, X_1))$$

Conditions:

1. All of the aggregation expressions in $F^2_{X_1}$ reference only constants and the external names of $F^1_{X_1}$, and can be rewritten as equivalent aggregation expressions $F^{2\circ 1}_{X_1}$ which reference only constants and attributes in $\mathit{schema}(X_1)$.
2. All of the aggregation functions in $F_{\overline{\mathbf{X}}}$ are duplicate insensitive.
3. The aggregate values generated by $F^1_{X_1}$ are not used in any predicates or grouping sets. In other words, the external names in $F^1_{X_1}$ do not occur in ρ or α.

3.3.3 Partial Lazy Grouping

Rule `PU3` [9]
Assume that in the following query, $F_{\overline{\mathbf{X}}_2}$ and $F^1_{X_3}$ are sets of aggregation expressions whose internal expressions only reference constants as well as attributes from $\overline{\mathbf{X}}_2$ and X_3, respectively, and $F^2_{X_3}$ is a set of aggregation expression whose internal expressions only reference constants as well as attributes from $\mathit{external}(F^1_{X_3})$.

$$\Lambda^{\alpha}_{F_{\overline{\mathbf{X}}_2} \cup F^2_{X_3}}(\Pi^{\beta}_{\rho}(X_1, \overline{\mathbf{X}}_2, \Lambda^{\gamma}_{F^1_{X_3}}(X_3)))$$
$$\Downarrow$$
$$\Lambda^{\alpha}_{F_{\overline{\mathbf{X}}_2} \cup F^2_{X_3}}(\Pi^{\beta}_{\rho\setminus\rho'}(\overline{\mathbf{X}}_2, \Lambda^{\gamma\cup\delta}_{F^1_{X_3}}(\Pi^{\lambda\cup\delta}_{\rho'}(X_1, X_3))))$$

where ρ' is defined as the set of predicates in ρ that reference only attributes in $\mathit{schema}(X_1) \cup \mathit{schema}(X_3)$, and δ is any set of attribute names from $\mathit{schema}(X_1)$ that contains all of the necessary join and output attributes of X_1—that is, $\delta \supseteq (\beta \cup \mathit{attributes}(\rho \setminus \rho')) \cap \mathit{schema}(X_1)$.

Conditions:

1. ρ does not contain any predicates that reference external names from $F^1_{X_3}$.
2. All of the expressions in $F^2_{X_3}$ cleanly compose with the expressions in $F^1_{X_3}$ which they depend upon.
3. All of the aggregation expressions in $F_{\overline{\mathbf{X}}_2}$ are duplicate insensitive aggregation functions.

4 Query Rewriting Using Compensations

Algorithms that perform bottom-up view matching have been described by Zaharioudakis et al.[18] from IBM, as well as in patent applications by IBM[5] and Microsoft[10]. The algorithms from IBM take as input a query Q and a view definition V—both represented as QGM graphs—and return a rewriting of Q using V (if such a rewriting can be found). A rewriting is modeled as a new QGM graph that is equivalent to Q but contains a reference to V (by name, not by definition) as a leaf; all operators in the new graph besides V are referred to as the *compensation operators* required to make V equivalent to Q. As described by the authors, the algorithms operate at the logical rewrite phase (prior to cost-based optimization), and so because they only return one (or a small fixed number) of rewritings, presumably the algorithms make choices heuristically (since the components of the query have not yet been costed by the optimizer). The Microsoft algorithm is similar, but it functions during the cost-optimization stage. The advantage to this is that choices can be made based upon estimated cost, rather than heuristically. The disadvantage is that a cost-optimizer generates plans in which join ordering is already fixed, while view matching is independent of join order, meaning that some extra overhead occurs collapsing physical plans down to an unordered representation and testing whether the desired match has already been calculated for another equivalent join order.

Since the three algorithms mentioned above all work similarly, we will choose Zaharioudakis et al.'s as representative and will limit our discussion in the sequel to comparing against their algorithm.

4.1 The Bottom-up Matching Algorithm

Zaharioudakis et al.'s algorithm roughly works as follows: Given Q and V, assume that all combinations (with some limitations) of the child blocks Q_i and V_j of Q and V, respectively, have already been compared, and where a rewriting is possible the compensation operators needed to make V_j equivalent to Q_i have been calculated. Furthermore, suppose you are given a one-to-one mapping of successful rewritings between a subset of the child blocks of Q and a subset of the child blocks of V such that any V_j that does not have a matching Q_i in the mapping can be logically deleted from V using Rule `CR` (after any attributes of V_j have been removed from the output set of V). Zaharioudakis et al. then provide a set of templates that calculate the compensation operators for V from the compensation operators of its matched child blocks, the unmatched child blocks of Q, and any difference in the predicates, output attributes, or grouping attributes of Q and V.

Although the bottom-up template-based algorithm is undoubtedly efficient, the method described by Zaharioudakis et al. has a few fundamental weaknesses. First of all, the correctness of a given template is hard to verify. The templates given in [18] work by copying operators from within the given query Q and attaching them to the top of the view definitions V. The compensated view definition then replaces Q. Unfortunately, when the rewriting process is viewed this way, it is unclear how the copying should be done to preserve equivalence. For example, given a template that copies compensation operators from several subquery-subview matches to the top of V, how do we know that the chosen order in which to perform the copying preserves equivalence? Furthermore, how can we generate (correct) templates that implement rewritings whose correctness is not obvious? Those authors only give templates covering the simple (albeit most common) cases where the correctness of the rewrite can be intuited.

A second weakness of Zaharioudakis' algorithm is that it is prone to what we will call *multi-level failures*. A multi-level failure occurs when all of the non-logically-deletable relations and predicates in V have matching components somewhere in Q, but the bottom-up algorithm fails because the matching components occur at different heights in the tree. Example 2 illustrates a simple multi-level failure. The multi-level failure is caused by the restriction imposed by the bottom-up algorithm that a query block and a view block can only be matched if there is a one-to-one matching among their children (modulo the children in V that are logically-deletable and the children in Q that can be pushed up). This requirement of one-to-one child matching is reasonable—it limits the potential matches to explore to a tractable number—but it generates false negatives.

Example 2 Consider the student and enrollment tables from Example 1, and the following queries

Q^a and Q^b phrased over them.

$$Q^a := {}^{\mathtt{Q^a}}\prod^{\{\mathtt{sid,sname,grade,credits}\}}_{\{\mathtt{sid=student,sname=``Smith'',grade=A}\}}(\mathtt{S},\mathtt{E})$$
$$Q^b := {}^{\mathtt{Q^b}}\prod^{\{\mathtt{sid,sname,grade,credits}\}}_{\{\mathtt{sid=student,sname=``Smith''}\}}(\mathtt{S},\ {}^{\mathtt{Q^b_1}}\prod^{\{\mathtt{student,grade,credits}\}}_{\{\mathtt{grade=A}\}}(\mathtt{E}))$$

Suppose the Q^a is the user query, and Q^b is the view definition. A bottom-up matching algorithm will start by matching subview Q^b_1 with query Q^a and determine that this intermediate match is successful with compensation involving a join of relation S. However, the overall match will then fail because the root of the user query has been reached while unmatched levels remain in the view. Conversely, if Q^b is the user query and Q^a is the view, then the bottom-up algorithm attempts to match subquery Q^b_1 with view Q^a but fails because Q^a contains a non-logically-deletable child ‘‘S’’, which cannot be matched against any child of Q^b_1.

Quite obviously the failures in this example could be avoided by first flattening the definition of Q^b using Rule VF1. However, when the different blocks in the query definitions contain aggregation, there may be no such obvious normalization strategy that will avoid multi-level failures.

4.2 A Generalized Subsumption Algorithm Using Compensations

The following algorithm is a generalization of the template-based algorithm described by Zaharioudakis et al. A few significant modifications have been made to the algorithm:

1. Rather than using templates that copy operators from the query to the view compensation, the new algorithm uses an extensible set of transformation rules to iteratively re-shape the original query definition into a query containing the view definition as a subtree (which means the ancestors to the subtree form the compensation). Using this iterative re-shaping approach has two advantages over templates:
   (a) The correctness of any rewrite can be justified by citing the sequence of transformation rules that generated it.
   (b) Adding a new transformation rule to the set of available transforms automatically increases the power of the rewriting algorithm.
2. The source of multi-level failures mentioned in the previous section has been partially avoided, for the case where the view has more levels than the equivalent query. The novel logic is to allow the root query operator of Q to match against *both* the root of V *and* one of its children V_j. Given the queries Q^a and Q^b from Example 2, our modified algorithm would successfully rewrite query Q^a to use view Q^b (in contrast to the failure of the original algorithm), but it would still fail to rewrite Q^b to use view Q^a.
3. The algorithm has been written to navigate the query and view definitions in a top-down rather than bottom-up fashion for reasons that will be explained later in the paper. Although it is not explicitly written into the algorithm, a memo structure would obviously be used to eliminate redundant work caused by repeated intermediate invocations. It is important to note that even though the algorithm navigates the query and view definitons top-down, the matching process itself still proceeds essentially bottom-up, as (almost) all non-navigational work is performed after the recursive calls on the children have returned.

Note that as written, our algorithm makes non-deterministic choices. This emphasizes the fact that many different rewritings may be possible, depending upon the choices made; therefore, an implementation would need to either heuristically make decisions to obtain a single rewrite, or potentially explore all paths through the algorithm to obtain an optimal-cost solution. Because of the non-determinism, when the algorithm returns “FAIL”, it only indicates that no rewriting can be found *for the choices made so far by the algorithm*; it does not preclude that a different set of choices could find a correct rewriting.

Given an invocation of Match(Q, V), there are three different styles of matches that the algorithm can choose to make. The first match style recursively matches V completely within a subquery Q_i; this match style is an artifact of the top-down structuring of the algorithm and plays a navigational role analogous to the bottom-up "navigation function" in [18]. The second match style matches the root operator of Q against the root operator of V, and children of Q against children of V; this corresponds to the style of matches made by the templates in [18]. Finally, the third match style matches the root of Q against both a subview V_i and the root of V; this is the new match style mentioned above that was introduced to avoid multi-level failures for the case where the view has more levels than the matching query.

Assume that query Q and the definition of view V have been simplified (recursively) as much as possible using the following steps.

1. Adjacent Π operators (i.e. Π operators with Π children) are flattened using Rule VF1.

2. The heads of each Π and Λ subquery have been pruned using Rules AR1, AR2, and AR3, respectively.

3. Predicates are pushed down as far as possible without introducing adjacent Π operators.

Also assume that each base relation occurs at most once anywhere within a query or view definition (i.e. no self-joins), and that all attribute names are unique. These assumptions are only for simplicity of presentation; they can easily be removed by creating extra mappings for variables names and relation names.

The Matching Algorithm

```
Function MATCH($Q$, $V$) returns a rewriting of $Q$ using $V$, or FAIL
   Guess a match style of 1, 2, or 3.

   Case 1:  // Pass-thru invocation to subquery of $Q$
      Guess a subquery $Q_i$ and recursively call MATCH($Q_i$, $V$) returning $C^{Q_i,V}$.
      If $C^{Q_i,V}$ == FAIL, return FAIL.
      Otherwise, replace $Q_i$ with $C^{Q_i,V}$, and return the modified $Q$.

   Case 2:  // Match of root operators of $Q$ and $V$
      If the root operators of $Q$ and $V$ are not the same type, return FAIL.
      If the root operators of $Q$ and $V$ are both $\Pi$ then:
         Let $\Pi_Q$, $\Pi_V$ denote the initial root operators of $Q$,$V$ with
               output sets $\alpha_Q$, $\alpha_V$ and predicate sets $\rho_Q$, $\rho_V$.
               // $\alpha_Q$ and $\alpha_V$ are only initial root operators for
               // $Q$ and $V$ because as we use transformations to push other operators
               // above them, the definitions of $Q$ and $V$ will become rooted by
               // other operators.
         For each pair of relations $R_i^Q$,$R_j^V$ listed in $\Pi_Q$ and $\Pi_V$, respectively,
               such that $R_i^Q$ and $R_j^V$ refer to the same relational schema:
            Mark $R_i^Q$ and $R_j^V$ as matched.
         For each pair of subqueries $Q_i$,$V_j$ in $\Pi_Q$ and $\Pi_V$, respectively, containing
               at least one common relation:
            Call MATCH($Q_i$, $V_j$) returning $C^{Q_i,V_j}$.
         Guess a partial one-to-one mapping $\theta : \{$subqueries of $\Pi_Q\} \rightarrow \{$subviews of $\Pi_V\}$
               such that $C^{Q_i,\theta(Q_i)} \neq$ FAIL.
         Mark each subquery $Q_i$ in the domain of $\theta$ as matched.
         Mark each subview $V_i$ in the range of $\theta$ as matched.
         For each unmarked relation or subquery $X$ in $\Pi_Q$:
            Push $X$ above $\Pi_Q$ using rule VF1.
            If $X$ cannot be pushed above $\Pi_Q$, return FAIL.
         Attempt to find an ordering of the unmarked children (relations and subviews)
               in $\Pi_V$ such that each unmarked child can be logically deleted from $\Pi_V$
               using rule CR (after first removing any of the child's attributes from
               the output set of $\Pi_V$).  If no such ordering can be found, return FAIL.
         Replace each subquery $Q_i$ in $\Pi_Q$ with its rewrite $C^{Q_i,\theta(Q_i)}$.
         While child compensation operators exist below $\Pi_Q$:
            Choose a child compensation operator immediately below $\Pi_Q$ and attempt to
                  find a transformation that will move it above $\Pi_Q$.
            If no such operator-transformation pair can be found, return FAIL.

         // At this point, $\Pi_Q$ and $\Pi_V$ now contain the exact same children.
         If $\rho_V \div \rho_Q$ is not empty[5], return FAIL.
         Insert a new operator $\Pi'_Q$ immediately above $\Pi_Q$ with output set $\alpha_Q$,
               $\Pi_Q$ as its only child, and $\rho_Q \div \rho_V$ as its predicate set.
         If $\alpha_Q \subseteq \alpha_V$, then
            Replace $\Pi_Q$ with a reference to $V$.
            Return $Q$ (where $Q$ now refers to the root of the rewritten query,
                  including the compensation operators).

         For each attribute $a \in \alpha_Q \setminus \alpha_V$
            If $\Sigma \cup \Delta$ plus predicates in $V$ imply that $\alpha_V \rightarrow a$, then
               // Attempt to rejoin the source of attribute $a$
               Let $X_a$ be lowest subquery or relation in the original $Q$ such that
                     $a \in schema(X_a)$ and there exists two ordered sets of attributes
                     $A \subseteq schema(X_a)$ and $B \subseteq \alpha_V$ such that $A \rightarrow Id_{X_a}$
                     and the predicates in $V$ imply that $A = B$.
               If no such $X_a$ exists, return FAIL.
               If $X_a$ is not already a child of $\Pi'_Q$, then
                  Add $X_a$ as a child of $\Pi'_Q$, and add $A = B$ to the
                        predicate set of $\Pi'_Q$.
         Replace $\Pi_Q$ with a reference to $V$.
         Return $Q$.

      Else, if the root operators of $Q$ and $V$ are both $\Lambda$ then:
         Let $\Lambda_Q$, $\Lambda_V$ denote the initial root operators of $Q$,$V$ with
               grouping sets $\alpha_Q$, $\alpha_V$ and aggregation expressions $F_Q$, $F_V$.
         Call MATCH($Q_1$, $V_1$) returning $C^{Q_1,V_1}$.
         If $C^{Q_1,V_1}$ == FAIL, then return FAIL.
         Replace subquery $Q_1$ in $\Lambda_Q$ with its rewrite $C^{Q_1,V_1}$.
         While child compensation operators exist below $\Lambda_Q$:
            Attempt to find a transformation that will push the nearest
                  child compensation operator above $\Lambda_Q$.
            If no such transformation can be found, return FAIL.

         // Now $\Lambda_Q$ has been rewritten to have the child $V_1$.
         If $\alpha_Q = \alpha_V$ then
            If $F_Q \subseteq F_V$, then
               // We (unrealistically) assume that equivalent aggregation expressions
```

```
                // in $\Lambda_Q$ and $\Lambda_V$ have identical external attribute names.
                // This can be improved by allowing expressions and variable
                // renaming in $\Pi$ operators.
                Replace $\Lambda_Q$ with a reference to $V$.
                Return $Q$.
             Otherwise return FAIL.
          Else
             // Re-aggregation will be needed.
             Insert a new operator $\Lambda'_Q$ immediately above $\Lambda_Q$ with grouping set $\alpha_Q$
                   and an initially empty aggregation expression set.
             For each aggregation expression $f(x)$ in $F_Q$:
                Attempt to rewrite $f(x)$ as $f'(C)$, where $C \in F_V$.
                If this succeeds, add $f'(C)$ to the aggregation expression set of $\Lambda'_Q$.
                Otherwise, return FAIL.
             If $\alpha_Q \subset \alpha_V$, then
                Replace $\Lambda_Q$ with a reference to $V$.
                Return $Q$.
             Else
                // Rejoin compensation will be needed to retrieve missing grouping attributes.
                Insert a new operator $\Pi'_Q$ between $\Lambda'_Q$ and $\Lambda_Q$, with
                      output attributes $\alpha_Q \cup external(F_V)$ and an empty predicate set.
                For each attribute $a \in \alpha_Q \setminus \alpha_V$:
                   If $\Sigma \cup \Delta$ plus predicates in $V$ imply that $\alpha_V \rightarrow a$, then
                      // Attempt to rejoin the source of attribute $a$
                      Let $X_a$ be lowest subquery or relation in the original $Q$
                            such that $a \in schema(X_a)$
                            and there exists two ordered sets of attributes
                            $A \subseteq schema(X_a)$ and $B \subseteq \alpha_V$ such that $A \rightarrow Id_{X_a}$
                            and the predicates in $V$ imply that $A = B$.
                      If no such $X_a$ exists, return FAIL.
                      If $X_a$ is not already a child of $\Pi'_Q$, then
                         Add $X_a$ as a child of $\Pi'_Q$, and add $A = B$ to the
                            predicate set of $\Pi'_Q$.
                Replace $\Lambda_Q$ with a reference to $V$.
                Return $Q$.

    Case 3:  // Match $Q$ first with a subview $V_j$, then with the root of $V$
       Guess a subview $V_j$ and recursively call MATCH($Q$, $V_j$) returning $C^{Q,V_j}$.
       If $C^{Q,V_j}$ == FAIL, then return FAIL.
       Else, return MATCH($C^{Q,V_j}$, $V$).
          // Within this recursive invocation, treat $V_j$ as a base relation within
          // both $C^{Q,V_j}$ and $V$.  In other words, do not recurse into the
          // definition of $V_j$, or else the recursion will not terminate.

End Function
```

Example 3 Suppose that you have a database tracking sales of a product using the following relational schemas:

[5]We use $\rho_1 \div \rho_2$ to denote the set of predicates in ρ_1 not implied by ρ_2 with respect to any schema constraints.

`R(Rid, Rname, Rman, Rpop)`	Region id, name, manager, and population
`S(Sid, Sname, Sregion)`	Store id, name, and region
`E(Eid, Ename, Estore)`	Employee id, name, and store
`C(Cid, Cname, Cage)`	Customer id, name, and age
`P(Pid, Pname, Pcat, Pcost)`	Product id, name, category, and cost
`ST(STstore, STcust, STprod, STdate, STprice)`	Sales Transaction

For each relation R_i, Σ contains the FD $\{Id_{R_i}\} \rightarrow schema(R_i)$. In addition, Σ also contains the following FDs:

$$\{\mathtt{Rid}\} \rightarrow Id_{\mathtt{R}}, \{\mathtt{Sid}\} \rightarrow Id_{\mathtt{S}}, \{\mathtt{Eid}\} \rightarrow Id_{\mathtt{E}}, \{\mathtt{Cid}\} \rightarrow Id_{\mathtt{C}}, \{\mathtt{Pid}\} \rightarrow Id_{\mathtt{P}}$$

As well, Δ contains the following inclusion dependencies:

$$\mathtt{S(Sregion)} \subseteq \mathtt{R(Rid)},\ \mathtt{E(Estore)} \subseteq \mathtt{S(Sid)},\ \mathtt{ST(STstore)} \subseteq \mathtt{S(Sid)},$$
$$\mathtt{ST(STcust)} \subseteq \mathtt{C(Cid)},\ \mathtt{ST(STprod)} \subseteq \mathtt{P(Pid)}$$

Suppose that the database contains the following view V recording sum of sales by region, which first calculates as a subview the sum of sales by store.

$$\mathtt{V} := {}^{\mathtt{V}}\Lambda^{\{\mathtt{Rid,Rname}\}}_{\{\mathtt{SUM(sales)\ as\ sales}\}}({}^{\mathtt{V_1}}\Pi^{\{\mathtt{Rid,Rname,sales}\}}_{\{\mathtt{Rid=Sregion}\}}(\mathtt{R}, {}^{\mathtt{V_{1.1}}}\Lambda^{\{\mathtt{Sid,Sname,Sregion}\}}_{\{\mathtt{SUM(STprice)\ as\ sales}\}}({}^{\mathtt{V_{1.1.1}}}\Pi^{\{\mathtt{Sid,Sname,Sregion,STprice}\}}_{\{\mathtt{STstore=Sid}\}}(\mathtt{S},\mathtt{ST}))))$$

Now suppose that the following query Q is posed to the database.

$$\mathtt{Q} := {}^{\mathtt{Q}}\Lambda^{\{\mathtt{Rname}\}}_{\{\mathtt{SUM(STprice)\ as\ sales}\}}({}^{\mathtt{Q_1}}\Pi^{\{\mathtt{Rname,STprice}\}}_{\{\mathtt{Rid=Sregion,Sid=STstore,Rname=``Canada"}\}}(\mathtt{R},\mathtt{S},\mathtt{ST}))$$

Algorithm `MATCH` can find a rewrite of Q to use V, as illustrated by the following trace of the algorithm:

```
Call MATCH($Q$, $V$)
   Choose match style 3
   Call MATCH($Q$, $V_1$)
      Choose match style 3
      Call MATCH($Q$, $V_{1.1}$)
         Choose match style 2
         Call MATCH($Q_1$, $V_{1.1.1}$)
            Choose match style 2
            Mark matched pairs of relations for $S$ and $ST$.
            Push $R$ above $\Pi_{Q_1}$ using rule VF1
         $C^{Q_1,V_{1.1.1}}$ := ${}^{\mathtt{C_1^{Q_1,V_{1.1.1}}}}\Pi^{\{\mathtt{Rname,STprice}\}}_{\{\mathtt{Rid=Sregion,Rname=``Canada"}\}}(\mathtt{R}, \mathtt{V_{1.1.1}})$
         Replace $Q_1$ with $C^{Q_1,V_{1.1.1}}$
         Push $C_1^{Q_1,V_{1.1.1}}$ above $Q$ using rule PD1
      $C^{Q,V_{1.1}}$ := ${}^{\mathtt{C_2^{Q,V_{1.1}}}}\Pi^{\{\mathtt{Rname,sales}\}}_{\{\mathtt{Rid=Sregion,Rname=``Canada"}\}}(\mathtt{R}, {}^{\mathtt{C_1^{Q,V_{1.1}}}}\Lambda^{\{\mathtt{Sregion}\}}_{\{\mathtt{SUM(sales)\ as\ sales}\}}(\mathtt{V_{1.1}}))$
      Call MATCH($C^{Q,V_{1.1}}$, $V_1$)
         Choose match style 2
         Mark matched pairs of relations for $R$.  Call MATCH($C_1^{Q,V_{1.1}}$, $V_{1.1}$)
            Choose match style 1
            Call to MATCH($V_{1.1}$, $V_{1.1}$) returns exact match
         $C^{C_1^{Q,V_{1.1}},V_{1.1}}$ := ${}^{\mathtt{C_1^{C_1^{Q,V_{1.1}},V_{1.1}}}}\Lambda^{\{\mathtt{Sregion}\}}_{\{\mathtt{SUM(sales)\ as\ sales}\}}(\mathtt{V_{1.1}})$
      Push $C_1^{C_1^{Q,V_{1.1}},V_{1.1}}$ above $C^{Q,V_{1.1}}$ using rule PU1
      $C^{C^{Q,V_{1.1}},V_1}$ := ${}^{\mathtt{C_2^{C^{Q,V_{1.1}},V_1}}}\Pi^{\{\mathtt{Rname,sales}\}}_{\{\mathtt{Rname=``Canada"}\}}({}^{\mathtt{C_1^{C^{Q,V_{1.1}},V_1}}}\Lambda^{\{\mathtt{Rid,Rname}\}}_{\{\mathtt{SUM(sales)\ as\ sales}\}}(\mathtt{V_1}))$
      Return $C^{C^{Q,V_{1.1}},V_1}$
   $C^{Q,V_1}$ := $\mathtt{C_2^{Q,V_1}}\Pi^{\{\mathtt{Rname,sales}\}}_{\{\mathtt{Rname=``Canada''}\}}(\mathtt{C_1^{Q,V_1}}\Lambda^{\{\mathtt{Rid,Rname}\}}_{\{\mathtt{SUM(sales)\ as\ sales}\}}(\mathtt{V_1}))$
   Call MATCH($C^{Q,V_1}$, $V$)
      Choose match style 1
      Call MATCH($C_1^{Q,V_1}$, $V$)
         Choose match style 2
         Call to MATCH($V_1$, $V_1$ returns exact match
         Exact match between grouping attributes and aggregation functions
      $C^{C_1^{Q,V_1},V}$ := $V$
   $C^{C^{Q,V_1},V}$ := $\mathtt{C_1^{C^{Q,V_1},V}}\Pi^{\{\mathtt{Rname,sales}\}}_{\{\mathtt{Rname=``Canada''}\}}(\mathtt{V})$
$C^{Q,V}$ := $\mathtt{C_1^{Q,V}}\Pi^{\{\mathtt{Rname,sales}\}}_{\{\mathtt{Rname=``Canada''}\}}(\mathtt{V})$
Return $C^{Q,V}$
```

Observe that V is partitioned into more levels that Q, which would cause the bottom-up algorithm of [18] to fail (even though each subquery in Q can be rewritten over a subview in V using only compensation operators). Our algorithm MATCH is able to find the rewrite only by using the third matching style which matches the root operator of the query against a subview.

5 Prerequisite Operators

Algorithm MATCH(Q,V) in Section 4 generates correct rewritings of query Q to use view V by utilizing a set of transformation rules, but it is far from complete. As mentioned earlier, one notable type of failure is when a query and an equivalent view are partitioned into a differing numbers of levels. Even though transformation rules may exist which can consolidate or partition levels (e.g. Rules PD1, PD2, etc.), we do not want the matching algorithm to have to exhaustively apply all possible transformations to either V or Q before a match can be found. The MATCH() algorithm of the previous section is an improvement over the original algorithm in that it finds rewritings where V is partitioned into more levels that Q (c.f. Example 3). However, the MATCH() algorithm will still fail when Q is partitioned into more levels that V. The following example illustrates the problem.

Example 4 Given the same schema and constraint information as in Example 3, consider the following query Q and view V (note that we've broken up the definition of Q at arbitrary points simply for readability).

$$Q := {}^{\mathtt{Q}}\Lambda^{\{\mathtt{Rname}\}}_{\{\mathtt{SUM(sales)\ as\ sales}\}}({}^{\mathtt{Q_1}}\Pi^{\{\mathtt{Rname,sales}\}}_{\{\mathtt{Rid=Sregion,Rname=``Canada''}\}}(\mathtt{R},\mathtt{Q_{1.1}}))$$
$$Q_{1.1} := {}^{\mathtt{Q_{1.1}}}\Lambda^{\{\mathtt{Sregion}\}}_{\{\mathtt{SUM(STprice)\ as\ sales}\}}({}^{\mathtt{Q_{1.1.1}}}\Pi^{\{\mathtt{Sregion,STprice}\}}_{\{\mathtt{STstore=Sid}\}}(\mathtt{S},\mathtt{ST}))$$

$$V := {}^{\mathtt{V}}\Lambda^{\{\mathtt{Rname}\}}_{\{\mathtt{SUM(STprice)\ as\ sales}\}}({}^{\mathtt{V_1}}\Pi^{\{\mathtt{Rname,STprice}\}}_{\{\mathtt{Rid=Sregion,Sid=STstore}\}}(\mathtt{R},\mathtt{S},\mathtt{ST}))$$

A successful bottom-up matching of Q to V requires that subquery $Q_{1.1.1}$ be matched against V_1. This matching will fail, as $Q_{1.1.1}$ cannot be rewritten to use V_1 due to the fact that V_1 contains a non-lossless join to relation R which does not occur within $Q_{1.1.1}$. This failure propagates upward, causing the matches that depend upon it to also fail. However,

$$\mathtt{Q} \equiv \Pi^{\{\mathtt{Rname,sales}\}}_{\{\mathtt{Rname=``Canada''}\}}(\mathtt{V})$$

with respect to $\Sigma \cup \Delta$, and so the matching algorithm has missed a valid rewriting opportunity.

A "prerequisite" is a condition that must be satisfied before a certain action can be taken. In Example 4, we would like to avoid the failure that occurs when attempting to rewrite $Q_{1.1.1}$ to use V_1; this failure could be avoided if we knew that higher up in the query tree, the result of $Q_{1.1.1}$ was going to be joined to relation `R` with the predicates `Rid=Sregion` and `Rname=''Canada''` (as is the case in query Q). If Q and V are matched bottom-up, then this information is not available; however, rather than declaring a failure, we could create a "specialized" solution that rewrites $Q_{1.1.1}$ to use V_1 and attaches to it a prerequisite condition stating that the join to `R` must occur higher in query Q in order for this rewrite to be valid. It is even possible that such a specialized solution would be preferable to a universal solution, if we had foreknowledge that the prerequisite conditions would satisfied. For example, if the join to `R` in V_1 was lossless, a universal rewrite of $Q_{1.1.1}$ to use V_1 would succeed by "logically deleting" `R` from V_1, but then within Q_1, `R` would be re-joined; however, a specialized solution that retained the necessary attributes from `R` that occur in V_1 and attached to the rewrite a prerequisite condition of a join to `R` could avoid the need to perform the later join specified within Q_1.

We extend the algebraic query language defined in Section 2.2 to include two new quasi-operators which embody the notion of a prerequisite. A prerequisite operator is not a full-fledged algebraic operator because it can not actually be evaluated. However, although prerequisite operators can not be evaluated (and therefore must not occur in the final query plan), they are useful to encode information passed between recursive invocations of the matching algorithm. As the rewriting process proceeds up the query tree, prerequisite operators are either resolved against query operators (and thus removed), or a transformation is needed to move a prerequisite of a child operator above its parent.

5.1 Prerequisite PSJ Operator

The Φ operator encapsulates prerequisite conditions on a rewritten query that can only be satisfied by an ancestor Π operator. It has the following form

$$\Phi_{\varphi}^{\delta;\ \epsilon}[\overline{\mathbf{X}}](Y)$$

which is made up of the following components.

- Y is a subquery expressed in the extended query algebra; it represents the rewritten query for which this Φ serves as a prerequisite. Y is considered to be this operator's single child operator.

- $\overline{\mathbf{X}}$ is a set of "extra" relations and/or subqueries expressed in the base (non-extended) query algebra; each $X_i \in \overline{\mathbf{X}}$ represents a relation/subquery occurring in the view definition which did not have a match in the original query.

- δ is a set of attributes "available" to be output by this operator; as such, it is required that δ be derivable from the child operator Y (i.e. $\delta \subseteq \mathit{schema}(Y)$).

- ϵ is a set of attributes "missing" from the output of this operator. In other words, ϵ contains all attributes in the schema of the original query which are not available from the rewritten query Y.

- φ is a set of "extra" predicates occurring in the view definition that were not implied by the predicates in the original query. This includes any predicates applied over $\mathit{schema}(\overline{\mathbf{X}})$ (including the join predicates). Note that it is required that predicates in φ can be applied over $\mathit{schema}(\Phi_{\varphi}^{\delta;\ \epsilon}[\overline{\mathbf{X}}](Y)) \cup \mathit{schema}(\overline{\mathbf{X}})$, or else there is no possibility that a matching predicate could be found higher up in the query.

Since a rewriting requires that all attributes in the schema of the original query be present in the schema of the rewritten query[6], it follows from the definitions of δ and ϵ that $\mathit{schema}(\cdot)$ is defined on Φ as

$$\mathit{schema}(\Phi_{\varphi}^{\delta;\ \epsilon}[\overline{\mathbf{X}}](Y)) := \delta \cup \epsilon$$

Example 5 Given the queries from Example 4, subquery $Q_{1.1.1}$ can be rewritten over view V_1 with conditions encapsulated by the following prerequisite operator.

$$\mathtt{Q}_{1.1.1} \equiv \Phi_{\{\mathtt{Rid=Sregion}\}}^{\{\mathtt{Rname,STprice}\};\ \{\mathtt{Sregion}\}}[\mathtt{R}](\mathtt{V}_1)$$

Observe that within the prerequisite operator, `Rname` is part of the available attribute set because it is available in V_1 and originates from the extra relation `R`; also, `Sregion` is listed as a missing attribute because it was in the original schema of $Q_{1.1.1}$ but is not available from V_1.

5.2 Prerequisite Aggregation Operator

The Ω operator encapsulates prerequisite conditions on a rewritten query that can only be satisfied by an ancestor Λ operator. It embodies an assumption made during a rewriting of the subquery beneath it that there would be an aggregation operator higher up. The Ω operator has the following form

$$\Omega_{F^a;\ F^m}^{\delta;\ \epsilon}(Y)$$

which is made up of the following components.

- Y is a subquery expressed in the extended query algebra; it represents the rewritten query for which this Ω serves as a prerequisite. Y is considered to be this operator's single child operator.
- δ is the set of attributes "available" as grouping attributes. In other words, δ specifies the maximum grouping granularity available to parent operators. It is required that δ be derivable from the child operator Y (i.e. $\delta \subseteq \mathit{schema}(Y)$).
- ϵ is a set of base-relation attributes "missing" from the output of this operator. In other words, ϵ contains all attributes in the schema of the original query that are not the result of an aggregation expression, and are not available from the rewritten query Y.
- F^a is a set of aggregation expressions "available" that were already calculated within Y. As such, it is required that $\mathit{external}(F^a) \subseteq \mathit{schema}(Y)$ (there are no requirements on $\mathit{attributes}(F^a)$). The parent operator would access the value of these expressions by their external attribute name, since they have already been calculated; the definition of the aggregation expression is stored so that the parent operator knows the origin of the attribute and is therefore able to use it for rewriting expressions.
- F^m is a set of aggregation expressions "missing" from the aggregation expressions available in Y. In other words, F^m contains all aggregation expressions calculated in the original query which are not available in the rewritten query Y. The definition of these expressions is stored to enable the parent operator to rewrite an expression that uses an external variable from F^m to instead use an external variable from one of the available expressions in F^a.

For reasons similar to the Φ operator, the function $\mathit{schema}(\cdot)$ is defined on Ω as

$$\mathit{schema}(\Omega_{F^a;\ F^m}^{\delta;\ \epsilon}(Y)) := \delta \cup \epsilon \cup \mathit{external}(F^a \cup F^m)$$

[6]One would normally expect a rewritten query to have a schema identical to the original query; however, we loosen this to a containment relationship in order to allow the output to contain attributes available from the set of extra relations.

Example 6 Given the queries from Example 4, subquery $Q_{1.1}$ can be rewritten over subview V with conditions encapsulated by the following prerequisite operators.

$$\mathtt{Q_{1.1}} \equiv \Phi^{\{\mathtt{Rname,Sregion,sales}\};\ \{\}}_{\{\mathtt{Rid=Sregion}\}}[\mathtt{R}](\Omega^{\{\mathtt{Rname}\};\ \{\mathtt{Sregion}\}}_{\{\mathtt{SUM(STprice)\ as\ sales}\};\ \{\}}(\mathtt{V}))$$

This solution can be interpreted as saying:

> $Q_{1.1}$ can be rewritten using V if when traversing the global query tree upwards starting at the root of $Q_{1.1}$, you see a Π operator that joins relation `R` with predicate `Rid=Sregion`, followed later by a Λ operator that does not include `Sregion` in the grouping set.

Some interesting points to observe:

- The attribute `Sregion` is listed within the set of "available" attributes by the Φ operator, even though `Sregion` is not available in view V. The knowledge that `Sregion` is missing is encapsulated within the Ω operator and is transparent to the operators above it. The parent operator of the Ω operator only needs to "become aware" that `Sregion` is not available when an attempt is made to either resolve the Ω operator against its parent or push the Ω operator past its parent.
- The attribute `Rname` which originates from an extra relation is included in the output schema of the new rewrite, even though it was not in the output schema of $Q_{1.1}$.

6 Transformations Involving Prerequisite Operators

This section presents transformation rules for simplifying, commuting, and removing prerequisite operators within an expression of the extended query algebra. For the duration of this section, let Y denote a subquery expressed in the extended query algebra, X a base relation or subquery expressed in the standard (non-extended) query algebra, $\overline{\mathbf{X}}$ a set of X, and $\overline{\mathbf{Y}}$ a set of Y. Note that besides the conditions listed limiting when a transformation rule can be correctly applied, there is also the expectation that the input and output operators in transform rule are syntactically valid.

6.1 Query Simplifying Transformations

6.1.1 Φ Introduction

This rule introduces or removes a trivial Φ operator that encapsulates no prerequisite conditions.

Rule `PI`

$$\Phi^{schema(Y);\ \emptyset}_{\emptyset}[\,](Y) \\ \Updownarrow \\ Y$$

Conditions: None.

6.1.2 Φ Flattening

This rule merges (or partitions) adjacent Φ operators.

Rule `PF`

$$\Phi^{\delta_1;\ \epsilon_1}_{\varphi_1}[\overline{\mathbf{X}}_1](\Phi^{\delta_2;\ \epsilon_2}_{\varphi_2}[\overline{\mathbf{X}}_2](Y)) \\ \Updownarrow \\ \Phi^{\delta_1 \setminus \epsilon_2;\ \epsilon_1 \cup \epsilon_2}_{\varphi_1 \cup \varphi_2}[\overline{\mathbf{X}}_1 \cup \overline{\mathbf{X}}_2](Y)$$

Conditions: None.

6.1.3 Missing Set Pruning

Theses rules remove unneeded attributes from the "missing" sets of a Φ or Ω operator.

Rule `MSP1`

$$\Phi_{\varphi}^{\delta;\ \epsilon}[\overline{\mathbf{X}}](Y)$$
$$\Updownarrow$$
$$\Phi_{\varphi}^{\delta;\ \epsilon\cap\gamma}[\overline{\mathbf{X}}](Y)$$

where γ is the set of attribute names mentioned anywhere within the parent operator of this Φ operator.

Conditions: none

Rule `MSP2`

$$\Omega_{F^a;\ F^m}^{\delta;\ \epsilon}(Y)$$
$$\Updownarrow$$
$$\Omega_{F^a;\ F^m\cap\gamma}^{\delta;\ \epsilon\cap\gamma}(Y)$$

where γ is the set of attribute names mentioned anywhere within the parent operator of this Ω operator.

Conditions: none

6.1.4 Rejoin Compensation Introduction

Like the rules in Section 6.1.3, this rule removes attributes from the missing set of a prerequisite operator; however, whereas Rules `MSP1` and `MSP2`only apply when the missing attributes are never used by their parent, these rules make the missing attributes available to their parent by introducing a child compensation operator that performs a join to retrieve the needed columns.

Rule `RCI1`

$$\Phi_{\varphi}^{\delta;\ \epsilon}[\overline{\mathbf{X}}](Y)$$
$$\Downarrow$$
$$\Phi_{\varphi}^{\delta\cup schema(X_1);\ \epsilon\setminus schema(X_1)}[\overline{\mathbf{X}}](\Pi_{\rho_{X_1}}^{\delta\cup schema(X_1)}(X_1, Y))$$

where X_1 is any relation or subquery occurring within the original query Q for which Y is a rewrite, and ρ_{X_1} is the set of predicates $\{a_i = b_i\}$, where the pair (a_i, b_i) is defined as below.

Conditions:

1. There exists two lists of attributes $A \subseteq schema(X_1)$ and $B \subseteq \delta$ such that
   (a) $\Sigma \cup \Delta$ implies that $A \to Id_{X_1}$; and
   (b) for each corresponding pair (a_i, b_i) such that a_i is the i-th element of A and b_i is the i-th element of B, either a_i and b_i are the same attribute name[7], or the predicates within the original query Q imply that $a_i = b_i$.

[7]Obviously, this condition relies on our earlier assumptions that all attribute names are unique, the original query contains no self-joins, and the query language does not include facilities for attribute renaming. Removing these assumptions is possible, but requires some addition variable-mapping notation that is orthogonal to the contributions of this paper.

Rule `RCI2`

$$\Omega^{\delta;\ \epsilon}_{F^a;\ F^m}(Y)$$
$$\Downarrow$$
$$\Omega^{\delta \cup schema(X_1);\ \epsilon \setminus schema(X_1)}_{F^a;\ F^m}(\Pi^{\delta \cup schema(X_1)}_{\rho_{X_1}}(X_1, Y))$$

where X_1 is any relation or subquery occurring within the original query Q for which Y is a rewrite, and ρ_{X_1} is the set of predicates $\{a_i = b_i\}$, where the pair (a_i, b_i) is defined as below.

Conditions:

1. There exists two lists of attributes $A \subseteq schema(X_1)$ and $B \subseteq \delta$ such that
   (a) $\Sigma \cup \Delta$ implies that $A \rightarrow Id_{X_1}$; and
   (b) for each corresponding pair (a_i, b_i) such that a_i is the i-th element of A and b_i is the i-th element of B, either a_i and b_i are the same attribute name[7], or the predicates within the original query Q imply that $a_i = b_i$.

6.1.5 Prerequisite Conjunct Removal

This rule removes conjuncts from the "extra" set which can be "logically deleted" from the underlying expression Y. This performs exactly the same function as Rule `CR`, except that Rule `CR` must be applied on the standard relational operators, while this rule allows the conjunct removal to be deferred until a later time and performed on the prerequisite operator instead.

Rule `PCR`

$$\Phi^{\delta;\ \epsilon}_{\varphi}[\overline{\mathbf{X}}, X_1](Y)$$
$$\Downarrow$$
$$\Phi^{\delta \setminus schema(X_1);\ \epsilon}_{\varphi \setminus \varphi'}[\overline{\mathbf{X}}](Y)$$

where φ' is the set of predicates in φ that reference attributes in $schema(X_1)$.

Conditions:

1. The (future) join to X_1 would be a lossless join. More formally, define A to be the sets of attribute names from X_1 appearing φ' (i.e. $A := attributes(\varphi') \cap schema(X_1)$), and B to be the remaining attributes in φ' (i.e. $B := attributes(\varphi') \setminus A$; note that syntactic correctness of the given Φ operator guarantees that $B \subseteq \delta \cup \epsilon$). Then,
   (a) All of the predicates in φ' are equality join predicates between an attribute in A and an attribute in B (i.e. no filtering predicates over only X_1); and
   (b) $\Sigma \cup \Delta$ plus any predicates in Y, X_1, and φ imply that
      i. $A \rightarrow Id_{X_1}$ and
      ii. $Y(B) \subseteq X_1(A)$.

6.2 Prerequisite Resolution

These rules remove some or all of the prerequisite conditions embodied with an operator by resolving them against an operator in the standard query algebra.

6.2.1 Φ Resolution

These rules resolve predicates and extra conjuncts in a Φ operator against components of an adjacent Π operator. Observe that if all of the predicates and extra conjuncts in a Φ operator are resolved and the missing attribute set is empty (possibly after applying rules from Sections 6.1.3 or 6.1.4), then the Φ operator can be subsequently removed using Rule `PI`.

Rule `PR1`

$$\Pi^{\alpha}_{\rho}(\overline{\mathbf{Y}}, X, \Phi^{\delta;\ \epsilon}_{\varphi}[\overline{\mathbf{X}}, X](Y))$$
$$\Downarrow$$
$$\Pi^{\alpha}_{\rho \div \varphi}(\overline{\mathbf{Y}}, \Phi^{\delta;\ \epsilon \cup \epsilon'}_{\varphi \div \rho}[\overline{\mathbf{X}}](Y))$$

where $A \div B$ refers to the set of predicates in A *not* implied by B, with respect to any available schema information $(\Sigma \cup \Delta)$ or other predicates within Y; and $\epsilon' := (\alpha \setminus \delta) \cap \mathit{schema}(X)$ is the set of attributes from X that are output in α, but not available from Y.

Conditions:

1. $\mathit{schema}(X) \cap \mathit{attributes}(\rho \div \varphi) \subseteq \delta$
   In other words, all of the attributes of $\mathit{schema}(X)$ which occur in predicates in $\rho \div \varphi$ are available from Y. If the transformation were performed when this condition is not satisfied, $\rho \div \varphi$ would contain predicates that can not be evaluated. Note that in order to satisfy this condition, ρ might first have to be rewritten to minimize references to attributes in $\mathit{schema}(X)$. (For example, if $x \in \mathit{schema}(X)$, then $\rho := \{x = y, x = 3\}$ should first be rewritten as $\rho := \{x = y, y = 3\}$.)

Rule `PR2`

$$\Phi^{\delta;\ \epsilon}_{\varphi}[\overline{\mathbf{X}}, X](\Pi^{\alpha}_{\rho}(\overline{\mathbf{Y}}, X, Y))$$
$$\Downarrow$$
$$\Phi^{\delta';\ \epsilon \cup (\delta \setminus \delta')}_{\varphi \div \rho}[\overline{\mathbf{X}}](\Pi^{\alpha'}_{\rho \div \varphi}(\overline{\mathbf{Y}}, Y))$$

where $A \div B$ defined as previously; $\alpha' := \alpha \cap (\mathit{schema}(\overline{\mathbf{Y}}) \cup \mathit{schema}(Y))$ is is the set of attributes from α that remain available after the join to X is removed; and $\delta' := \delta \cap \alpha'$ is the set of attributes from δ that remain available after the join to X is removed.

Conditions:

1. $\mathit{schema}(X) \cap \mathit{attributes}(\rho \div \varphi) \subseteq (\mathit{schema}(\overline{\mathbf{Y}}) \cup \mathit{schema}(Y))$
   In other words, all of the attributes of $\mathit{schema}(X)$ which occur in predicates in $\rho \div \varphi$ are available from $\overline{\mathbf{Y}}$ or Y.

6.2.2 Ω Resolution

These rules resolve a Ω operator against a parent Λ operator. Note that Ω-resolution differs from Φ-resolution in a fundamental way. A Φ operator records extra predicates and joins that *have already been performed* in its child subquery and therefore Φ-resolution need to *cancel out* the matching predicates and joins in the Π operator. In contrast, the Ω operator records aggregation that its child subquery *has assumed to exist* in order for an underlying transformation to be correct. Hence, Ω-resolution leaves the parent Λ operator intact.

Rule `PR3a`

$$\Lambda^{\alpha}_{F}(\Omega^{\delta;\ \epsilon}_{F^a;\ F^m}(Y))$$
$$\Downarrow$$
$$\Lambda^{\alpha}_{F_\delta \cup F_{F^a}}(Y)$$

Conditions:

1. $\alpha \subseteq \delta$
   In other words, all of the attributes needed for partitioning into aggregation groups are available from Y.

2. The set of aggregation expressions in F can be rewritten as two sets of expressions F_δ, F_{F^a} such that:
   (a) Each expression in F is equivalent to an expression in $F_\delta \cup F_{F^a}$ (with identical external attribute name).
   (b) $\mathit{attributes}(F_\delta) \subseteq \delta$ and all aggregation expressions in F_δ are duplicate insensitive.
   (c) $\mathit{attributes}(F_{F^a}) \subseteq \mathit{external}(F^a)$ and all expressions in F_{F^a} cleanly compose with the expressions in F^a on which they depend.

Observe that the first condition of Rule PR3a seems unnecessarily restrictive; if an attribute $a \in \alpha$ is not available in δ (therefore $a \in \epsilon$), but there another attribute in $b \in \delta$ such that the predicates in Y imply that $a = b$, then we would like to simply rewrite α to replace a with b. However, we have not endowed our query language with the ability to perform attribute renaming, so this rewrite rewrite is unsafe, as it would change the output schema of the aggregation operator. This can be solved by introducing a new Φ operator above the rewrite that records the fact that a is not available, but b is.

Rule PR3b

$$\Lambda^{\alpha}_{F}(\Omega^{\delta;\ \epsilon}_{F^a;\ F^m}(Y))$$
$$\Downarrow$$
$$\Phi^{\alpha' \cup \mathit{external}(F);\ \alpha \setminus \alpha'}_{\emptyset}[\,](\Lambda^{\alpha'}_{F_\delta \cup F_{F^a}}(Y))$$

where α' is formed from α by replacing the attributes in $\alpha \cap \epsilon$ with attributes in δ that can be reasoned to be equivalent by the predicates occurring in Y.
Conditions:

1. $\alpha' \subseteq \delta$ and $\alpha' \rightarrow \alpha$
2. Same as Condition 2 in Rule PR3a.

6.3 Pulling Prerequisite Operators above Standard Operators

6.3.1 Non-Aggregated Φ Pull-Up

The following rule pulls a Φ past a Π operator. It assumes that Rules PR1 and MSP1 have already been applied, so there are no more predicates and/or extra relations that can be resolved between the Π and Φ operators, and the missing set of Φ is minimized.

Rule PPU1

$$\Pi^{\alpha}_{\rho}(\overline{\mathbf{Y}}, \Phi^{\delta;\ \epsilon}_{\varphi}[\overline{\mathbf{X}}](Y))$$
$$\Downarrow$$
$$\Phi^{\alpha';\ \epsilon}_{\varphi}[\overline{\mathbf{X}}](\Pi^{\alpha'}_{\rho}(\overline{\mathbf{Y}}, Y))$$

where $\alpha' := (\delta \cup \mathit{schema}(\overline{\mathbf{Y}})) \cap (\alpha \cup \mathit{schema}(\overline{\mathbf{X}}))$. The intuition is that the output set should include all of the available attributes that were either originally output by the Π operator or originate from the extra relations $\overline{\mathbf{X}}$ that are available in Y.

Conditions:

1. $\mathit{attributes}(\rho) \subseteq \delta \cup \mathit{schema}(\overline{\mathbf{Y}})$
   In other words, the predicates in ρ (possibly after suitable rewriting) only act over attributes that are available in Y or $\overline{\mathbf{Y}}$.
2. $\mathit{attributes}(\varphi) \subseteq \alpha \cup \mathit{schema}(\overline{\mathbf{X}})$
   In other words, the predicates in φ (that need to be resolved against predicates in parent operators) only act over attributes either in the original output set or in the schemas of the extra relations.

6.3.2 Invariant Φ Pull-Up

The following rule pulls a Φ past a Λ operator by reasoning that the aggregation is invariant with respect to the extra relations recorded by Φ. Pulling a Φ operator up past a Λ operator is in a sense equivalent to pushing a Π operator down past the Λ operator, which is why the conditions are similar to Rule `PU1`.

Rule `PPU2`

$$\Lambda_F^{\alpha}(\Phi_{\varphi}^{\delta;\ \epsilon}[\overline{\mathbf{X}}](Y))$$
$$\Downarrow$$
$$\Phi_{\varphi}^{\alpha' \cup external(F);\ \epsilon}[\overline{\mathbf{X}}](\Lambda_F^{\alpha'}(Y))$$

where $\alpha' := \delta \cap (\alpha \cup schema(\overline{\mathbf{X}}))$.

Conditions:

1. $attributes(F) \subseteq \delta$
   That is, the attributes used in the aggregation expressions in F (possibly after suitable rewriting) are all available from Y.
2. $attributes(\varphi) \subseteq \alpha \cup schema(\overline{\mathbf{X}})$
   In other words, the predicates in φ (that need to be resolved against predicates in parent operators) only act over attributes either in the original grouping set or in the schemas of the extra relations.
3. $\Sigma \cup \Delta$ plus predicates in φ and within Y imply that $\alpha' \rightarrow Id_X$ for each $X \in \overline{\mathbf{X}}$.

6.3.3 Partial Lazy Φ Pull-Up

The following rule pulls a Φ past a Λ operator even when the grouping is not invariant with respect to the extra relations recorded by Φ. Because the grouping possibly varies due to this transformation, it forces the creation of an aggregate prerequisite operator. This aggregate prerequisite forces the existence of another aggregation operator higher up in order to resolve, and it is the existence of this higher up aggregation operator that then justifies the correctness of the rewrite (essentially by Rule `PU3`). The fact that the new Ω prerequisite operator is created below the Φ prerequisite operator forces the existence of a Π operator that resolves with the Φ *below* the ancestor Λ operator (i.e. due to the nature of the bottom-up resolution, the ordering of nested prerequisite operators is opposite that of the the standard operators against which they need to resolve).

Rule `PPU3`

$$\Lambda_F^{\alpha}(\Phi_{\varphi}^{\delta;\ \epsilon}[\overline{\mathbf{X}}](Y))$$
$$\Downarrow$$
$$\Phi_{\varphi}^{\alpha' \cup \epsilon \cup external(F);\ \emptyset}[\overline{\mathbf{X}}](\Omega_{F;\ \emptyset}^{\alpha';\ \epsilon}(\Lambda_F^{\alpha'}(Y)))$$

where $\alpha' := \delta \cap (\alpha \cup schema(\overline{\mathbf{X}}))$.

Conditions:

1. $attributes(F) \subseteq \delta$
   That is, the attributes used in the aggregation expressions in F (possibly after suitable rewriting) are all available from Y.
2. $attributes(\varphi) \subseteq \alpha \cup schema(\overline{\mathbf{X}})$
   In other words, the predicates in φ (that need to be resolved against predicates in parent operators) only act over attributes either in the original grouping set or in the schemas of the extra relations.

6.3.4 Ω Pull-Up

The following rule pulls a Ω prerequisite operator past a Π operator. Note that since Ω resolution requires that an ancestor operator will be an aggregation operator, we do not have to worry about the multiplicity of tuples in this transformation (since duplicate tuples will be aggregated away by the ancestor operator, and the conditions of Rule `PR3a` guarantee that the value of the aggregation expressions will not be affected).

Rule `PPU4`

$$\begin{gathered}\Pi_{\rho}^{\alpha}(\overline{\mathbf{X}}, \Omega_{F^a;\ F^m}^{\delta;\ \epsilon}(Y)) \\ \Downarrow \\ \Omega_{F^a;\ F^m}^{\alpha\cap(\delta\cup schema(\overline{\mathbf{X}}));\ \epsilon}(\Pi_{\rho}^{\alpha\cap(\delta\cup external(F^a)\cup schema(\overline{\mathbf{X}}))}(\overline{\mathbf{X}}, Y))\end{gathered}$$

Conditions:

1. $attributes(\rho) \cap external(F^a \cup F^m) = \emptyset$
   None of the predicates in ρ act over the intermediate results of the inner aggregation expressions. This is necessary because the values of these intermediate results may have been affected by the underlying rewrite that generated the aggregation prerequisite operator.
2. $attributes(\rho) \cap \epsilon = \emptyset$
   None of the predicates in ρ act over missing attributes.

6.4 Commuting Prerequisites

The rules in this section commute a pair of adjacent Φ and Ω operators. Note that since the Ω operator does not actually perform aggregation but instead stipulates that an aggregation operator exists as an ancestor, the correctness of this transformation does not depend upon whether or not the grouping is invariant to the join of the extra relations in the Φ operator.

Rule `CP1`

$$\begin{gathered}\Phi_{\varphi}^{\alpha;\ \beta}[\overline{\mathbf{X}}](\Omega_{F^a;\ F^m}^{\delta;\ \epsilon}(Y)) \\ \Downarrow \\ \Omega_{F^a;\ F^m}^{\delta\cup\beta;\ \epsilon}(\Phi_{\varphi}^{\delta\cup external(F^a);\ \beta}[Y, \overline{\mathbf{X}}]())\end{gathered}$$

Conditions: none.

Rule `CP2`

$$\begin{gathered}\Omega_{F^a;\ F^m}^{\delta;\ \epsilon}(\Phi_{\varphi}^{\alpha;\ \beta}[Y, \overline{\mathbf{X}}]()) \\ \Downarrow \\ \Phi_{\varphi}^{\alpha;\ \beta\cup\epsilon}[\overline{\mathbf{X}}](\Omega_{F^a;\ F^m}^{\delta\setminus\beta;\ \emptyset}(Y))\end{gathered}$$

Conditions:

1. $attributes(\varphi) \cup external(F^a \cup F^m) = \emptyset$
   None of the predicates refer to attributes that are the result of an aggregation expression.

7 Using Prerequisites for Calculating View Subsumption

This section describes an extension of the MATCH(Q,V) algorithm described in Section 4 to take advantage of prerequisite operators (Section 5) and transformation rules available for manipulating them (Section 6). As described in Section 5, the bottom-up matching process suffers from the restriction that a query Q can only be rewritten to use a view V if each descendant subquery in Q subsumes a descendant subview in V—leading to missed rewrite opportunities such as the one in in Example 4. The use of prerequisite operators can (at least partially) eliminate this restriction.

7.1 Extending the MATCH Algorithm

The matching logic of the new algorithm NEWMATCH(Q,V) proceeds similar to that of MATCH(Q,V), with the following crucial differences:

- Recursive calls may return rewrites that contain prerequisite operators. Assume that the operators in each returned rewrite are normalized so that the prerequisite operators are completely above the standard (compensation) operators.
- Before pulling up any child compensation operators or unmatched children of Q, the algorithm first uses transformations from Section 6 to pull all child prerequisite operators above Q—possibly resolving some prerequisites against Q in the process.
- NEWMATCH() does not automatically try to logically delete unmatched child subviews of V. Instead, the algorithm creates a prerequisite operator that lists each unmatched subview as an extra relation. If logical deletion is both possible and necessary, this will occur at a later stage when the prerequisite operator is simplified using Rule PCR.
- The algorithm does not automatically add compensation operators that rejoin relations or subqueries in order to obtain attributes output by Q that have been projected away in V. Instead, the algorithm creates a prerequisite operator that lists each of these attributes as "missing". If the attribute is truly required, the rejoin compensation will be added at a later stage when the prerequisite operator is simplified using Rules RCI1 or RCI2; otherwise the prerequisite operator will later be simplified using Rules MSP1 or MSP2which avoid the need for the rejoin.

The new algorithm presumes the same assumptions as the original regarding query and view definitions having been simplified and not containing self-joins.

The New Matching Algorithm

```
Function NEWMATCH($Q$, $V$)
   Returns either
      1.  A rewrite of $Q$ using $V$, possibly including prerequisite operators
         which are positioned above any compensation operators; or
      2.  FAIL

   Guess a match style of 1, 2, or 3.

   Case 1:  // Pass-thru invocation to subquery of $Q$
      Guess a subquery $Q_i$ and recursively call NEWMATCH($Q_i$, $V$) returning $C^{Q_i,V}$.
      If $C^{Q_i,V}$ == FAIL, return FAIL.
      Else
         // Treat the remainder of $Q$ as compensation operators,
         // and leave it up to the function SIMPLIFY-AND-NORMALIZE to
         // deal with any prerequisites present in $C^{Q_i,V}$.
         Replace $Q_i$ with $C^{Q_i,V}$
         Return SIMPLIFY-AND-NORMALIZE($Q$)

   Case 2:  // Match of root operators of $Q$ and $V$
      If the root operators of $Q$ and $V$ are not the same type, return FAIL.
      If the root operators of $Q$ and $V$ are both $\Pi$ then:
         Let $\Pi_Q$, $\Pi_V$ denote the root operators of $Q$,$V$ with
               attributes sets $\alpha_Q$, $\alpha_V$ and predicate sets $\rho_Q$, $\rho_V$.
         For each pair of relations $R_i^Q$,$R_j^V$ listed in $\Pi_Q$ and $\Pi_V$, respectively,
               such that $R_i^Q$ and $R_j^V$ refer to the same relational schema:
            Mark $R_i^Q$ and $R_j^V$ as matched.
         For each pair of subqueries $Q_i$,$V_j$ in $\Pi_Q$ and $\Pi_V$, respectively, containing
               at least one common relation:
            Call NEWMATCH($Q_i$, $V_j$) returning $C^{Q_i,V_j}$.
         Guess a partial one-to-one mapping $\theta : \{$subqueries of $\Pi_Q\} \rightarrow \{$subviews of $\Pi_V\}$
               such that $C^{Q_i,\theta(Q_i)} \neq$ FAIL.
         Mark each subquery $Q_i$ in the domain of $\theta$ as matched.
         Mark each subview $V_i$ in the range of $\theta$ as matched.
         Replace each subquery $Q_i$ in $\Pi_Q$ with its rewrite $C^{Q_i,\theta(Q_i)}$.
         While child prerequisite operators exist below $\Pi_Q$:
            Choose a child prerequisite operator immediately below $\Pi_Q$ and attempt to find
                  a sequence of transformations that will either resolve it with $\Pi_Q$
                  or move it above $\Pi_Q$ (note that this might require first
                  applying simplifying transformations such as Rule PCR).
            If no such operator-transformation pair can be found, return FAIL.
         For each unmarked relation or subquery $X$ in $\Pi_Q$:
            Push $X$ above $\Pi_Q$ using rule VF1.
            If $X$ cannot be pushed above $\Pi_Q$, return FAIL.
         While child compensation operators exist below $\Pi_Q$:
            Choose a child compensation operator immediately below $\Pi_Q$ and attempt to
                  find a transformation that will move it above $\Pi_Q$.
            If no such operator-transformation pair can be found, return FAIL.
         // Note that the above transformations potentially modify $\alpha_Q$ and $\rho_Q$.

         Use Rule PI to insert a trivial prerequisite operator $\Phi_Q$ immediately above $\Pi_Q$.
               // Note $\Phi_Q$ will have an available attribute set $\delta_Q := \alpha_Q$, a
               // missing attribute set $\epsilon_Q := \emptyset$, and an extra predicate set $\varphi_Q := \emptyset$.)
         For each unmarked relation or subview $X$ in $\Pi_V$:
            Add $X$ to $\Pi_Q$ and to the set of extra relations in $\Phi_Q$.
            Let $\alpha_X := \alpha_V \cap schema(X)$.
            Add $\alpha_X$ to both $\alpha_Q$ and $\delta_Q$.

         // $\Pi_Q$ and $\Pi_V$ now contain the same children
         For each predicate $p \in \rho_V \div \rho_Q$:
            Add $p$ to both $\rho_Q$ and the set of extra predicates in $\Phi_Q$.
         Insert a new operator $\Pi'_Q$ above $\Phi_Q$ with output set $\delta_Q$,
               $\Phi_Q$ as its only child, and $\rho_Q \div \rho_V$ as its predicate set.
               If $\rho_Q \div \rho_V$ acts on any attribute names not in $\delta_Q$,
               add these attribute names to the missing set $\epsilon_Q$ of $\Phi_Q$.
         For each attribute $a \in (\alpha_Q \setminus \alpha_V)$:
            Move $a$ from $\delta_Q$ to $\epsilon_Q$ within operator $\Phi_Q$.
         Replace $\Pi_Q$ with a reference to $V$.
         Return SIMPLIFY-AND-NORMALIZE($Q$)

      Else, if the root operators of $Q$ and $V$ are both $\Lambda$ then:
         Let $\Lambda_Q$, $\Lambda_V$ denote the root operators of $Q$,$V$ with initial
               grouping sets $\alpha_Q$, $\alpha_V$ and aggregation expressions $F_Q$, $F_V$.
         Call NEWMATCH($Q_1$, $V_1$) returning $C^{Q_1,V_1}$.
         If $C^{Q_1,V_1}$ == FAIL, then return FAIL.
         Replace subquery $Q_1$ in $\Lambda_Q$ with its rewrite $C^{Q_1,V_1}$.
         While child prerequisite operators exist below $\Lambda_Q$:
            Attempt to find a transformation that will either resolve the nearest
                  child prerequisite operator with $\Lambda_Q$ or will push it
                  above $\Lambda_Q$.
            If no such transformation can be found, return FAIL.
         While child compensation operators exist below $\Lambda_Q$:
            Attempt to find a transformation that will push the nearest
                  child compensation operator above $\Lambda_Q$.
            If no such transformation can be found, return FAIL.
         // Note that the above transformations potentially modify $\alpha_Q$ and $F_Q$.

         // Now $\Lambda_Q$ has been rewritten to have the child $V_1$
         If $\alpha_Q = \alpha_V$ then
            If $F_Q \subseteq F_V$ then
               // We (unrealistically) assume that equivalent aggregation expressions
               // in $\Lambda_Q$ and $\Lambda_V$ have identical external attribute names.
               // This can be improved by allowing expressions and variable
               // renaming in $\Pi$ operators.
               Replace $\Lambda_Q$ with a reference to $V$.
               Return SIMPLIFY-AND-NORMALIZE($Q$).
            Otherwise return FAIL.

         Elseif $\alpha_Q \subset \alpha_V$ then
            // Re-aggregation is needed.
            Insert a new operator $\Lambda'_Q$ immediately above $\Lambda_Q$ with grouping set $\alpha_Q$
                  and an initially empty aggregation expression set.
            For each aggregation expression $f(x)$ in $F_Q$:
               Attempt to rewrite $f(x)$ as $f'(C)$, where $C \in F_V$.
               If this succeeds, add $f'(C)$ to the aggregation expression set of $\Lambda'_Q$.
               Otherwise, return FAIL.
            Replace $\Lambda_Q$ with a reference to $V$.
            Return SIMPLIFY-AND-NORMALIZE($Q$).

         Elseif $\alpha_V \subset \alpha_Q$ then
            // Needed columns are missing, but this could potentially be fixed by rejoins.
            // Add a prerequisite operator; any rejoins can be added via Rule RCI2
            // when simplifying the prereq at a later time.
            Insert a new operator $\Omega'_Q$ immediately above $\Lambda_Q$ with the available
                  grouping set equal to $\alpha_V$, the missing grouping set equal
                  to $\alpha_Q \setminus \alpha_V$, the available aggregation expressions
                  equal to $F_V$, and the missing aggregation expressions equal to $F_Q \setminus F_V$.
            Replace $\Lambda_Q$ with a reference to $V$.
            Return SIMPLIFY-AND-NORMALIZE($Q$).

         Else
```

```
          // Re-aggregation is needed to remove the extra grouping attributes in $\alpha_V$,
          // but an aggregation prerequisite is also needed to account for the missing
          // attributes found in $\alpha_Q$ but not $\alpha_V$.
          Insert a new operator $\Omega'_Q$ immediately above $\Lambda_Q$ with the available
                grouping set equal to $\alpha_V \cap \alpha_Q$, the missing grouping set equal
                to $\alpha_Q \setminus \alpha_V$, the available aggregation expressions
                equal to $F_V$, and the missing aggregation expressions equal to $F_Q \setminus F_V$.
          Insert a new operator $\Lambda'_Q$ between $\Omega'_Q$ and $\Lambda_Q$ with
                grouping set $\alpha_Q \cup \alpha_V$
                and an initially empty aggregation expression set.
          For each aggregation expression $f(x)$ in $F_Q$:
             Attempt to rewrite $f(x)$ as $f'(C)$, where $C \in F_V$.
             If this succeeds, add $f'(C)$ to the aggregation expression set of $\Lambda'_Q$.
             Otherwise, return FAIL.
          Replace $\Lambda_Q$ with a reference to $V$.
          Return SIMPLIFY-AND-NORMALIZE($Q$).

   Case 3:  // Match $Q$ first with a subview $V_j$, then with the root of $V$
      Guess a subview $V_j$ and recursively call NEWMATCH($Q$, $V_j$) returning $C^{Q,V_j}$.
      If $C^{Q,V_j}$ == FAIL, return FAIL.
      Else
         Let $C'$ be the topmost non-prerequisite operator in $C^{Q,V_j}$.
         Call NEWMATCH($C'$, $V$) returning $C^{C',V}$.
            // Within this recursive invocation, treat $V_j$ as a base relation within
            // both $C'$ and $V$.  In other words, do not recurse into the
            // definition of $V_j$, or else the recursion will not terminate.
         If $C^{C',V}$ == FAIL, then Return FAIL.
         Substitute $C^{C',V}$ for $C'$ within $C^{Q,V_j}$.
         Return SIMPLIFY-AND-NORMALIZE($C^{Q,V_j}$)

End Function
```

As its name implies, the function SIMPLIFY-AND-NORMALIZE(Q) both simplifies and normalizes a (rewritten) query expression that potentially contains prerequisite operators. The goal of the simplification is to remove as many prerequisites as possible. The goal of the normalization is to leave the query in a useful normal form, which in this case means to leave all prerequisite operators above the compensation operators. Due to how the matching algorithm first pulls up all of the child prerequisites operators before the child compensation, it is possible that a Φ operator pulled up from one child contains an extra relation that could resolve against a rejoin within a Π compensation operator from another child (this scenario is illustrated in Example 7). A method for removing these unnecessary prerequisites is as follows.

1. Use transformations from Sections 6.2 and 6.3 to push all prerequisites to the top of the query. If this is not possible, then return FAIL.

2. Use Rules CP1 and PF to merge all Φ prerequisite operators into a single operator located immediately above the compensation operators.

3. Use transformations from Section 3.2 to pull up rejoined subqueries and/or relations wherever possible.

4. Use Rule PR2 to resolve the rejoins at the top of the compensation with the Φ operator.

This method leaves the query expression in the required normal form.

Example 7 This example demonstrates resolution of a Φ prerequisite operator from one subquery with a Π compensation operator from a sibling subquery.

Given the schema and constraint information from Example 3, consider the following query Q and view V, which both calculate the number of employees and total sales for each store with an id greater than 100.

$$Q :=^{\mathtt{Q}}\Pi^{\mathtt{Sid,Sname,ecount,sales}}_{\mathtt{Sid=STstore}}(\mathtt{Q_1}, \mathtt{Q_2})$$
$$Q_1 :=^{\mathtt{Q_1}}\Lambda^{\mathtt{STstore}}_{\mathtt{SUM(STprice)\ as\ sales}}(\mathtt{ST})$$
$$Q_2 :=^{\mathtt{Q_2}}\Lambda^{\mathtt{Sid,Sname}}_{\mathtt{COUNT(Eid)\ as\ ecount}}(^{\mathtt{Q_{2.1}}}\Pi^{\mathtt{Sid,Sname,Eid}}_{\mathtt{Estore=Sid,Estore>100}}(\mathtt{E}, \mathtt{S}))$$

$$V :=^{\mathtt{V}}\Pi^{\mathtt{Sid,Sname,ecount,sales}}_{\mathtt{Sid=Estore}}(\mathtt{V_1}, \mathtt{V_2})$$
$$V_1 :=^{\mathtt{V_1}}\Lambda^{\mathtt{Sid,Sname}}_{\mathtt{SUM(STprice)\ as\ sales}}(^{\mathtt{V_{1.1}}}\Pi^{\mathtt{Sid,Sname,STprice}}_{\mathtt{Sid=STstore,STstore>100}}(\mathtt{S}, \mathtt{ST}))$$
$$V_2 :=^{\mathtt{V_2}}\Lambda^{\mathtt{Estore}}_{\mathtt{COUNT(Eid)\ as\ ecount}}(\mathtt{S})$$

The match of Q_1 with V_1 generates a prerequisite operator Φ containing an extra relation `‘‘S’’` and the predicate `‘‘Sid=STstore, STstore>100’’`, which will get pulled above operator Q using Rule `PPU1`. The match of Q_2 with V_2 generates a compensation operator Π that performs a join of relation `‘‘S’’` to V_2 using predicate `‘‘Estore=Sid, Estore>100’’`; this compensation operator can be pulled above operator Q using two applications of Rule `VF1`. The prerequisite operator can now resolve completely against the modified compensation operator below it using Rule `PR2` (since predicates in Q imply that `‘‘Estore=STstore=Sid’’`). The final rewrite can be simplified to $Q := V$.

7.2 Demonstrating the Value of Using Prerequisites

This section presents a large example which demonstrates how the `NEWMATCH` algorithm uses a combination of prerequisite and compensation operators in rewriting a complex, multi-block aggregation query to use a complex, multi-block aggregation view. Because certain subviews within the example cannot be subsumed by the subquery that they are matched against without the use of prerequisite operators, this illustrates a scenario where any algorithm that only uses compensation operators to perform bottom-up matching necessarily fails to find the rewrite.

Example 8 Suppose that the set of relational schemas from Example 3 is extended with an additional relation that represents a date dimension:

`D(Ddate, Dday, Dmonth, Dyear, Deom)`	Date, day, month, year, end-of-month flag

The field `Deom` is a flag indicating the last business day of each month, and is constrained (by logic outside of the database engine) so that exactly one date in each month is set to “Y”. Within the database, the schema constraints relevant to `D` include the following functional dependencies in Σ

$$\{Id_{\mathtt{D}}\} \rightarrow schema(\mathtt{D}), \{\mathtt{Ddate}\} \rightarrow Id_{\mathtt{D}}$$

and the following inclusion dependencies in Δ

$$\mathtt{ST(STdate)} \subseteq \mathtt{D(Ddate)}$$

The user wishes to generate a list showing for each year the cumulative yearly gross sales in each product category at each month's end. Suppose that the database already contains a logical view $Q_{1.1}$ which provides this information for each product.

$$Q_{1.1} := {}^{\mathtt{Q_{1.1}}}\Lambda^{\{\mathtt{Dmonth,Dyear,STprod}\}}_{\{\mathtt{SUM(STprice)\ as\ gross}\}}({}^{\mathtt{Q_{1.1.1}}}\Pi^{\{\mathtt{Dmonth,Dyear,STprod,STprice}\}}_{\{\mathtt{STdate{\leq}Ddate,year(STdate){=}Dyear,Deom{=}Y}\}}(\mathtt{ST},\mathtt{D}))$$

Then, one natural way that the user might choose to express this query is by joining the product table to the logical view, and then aggregating, which is represented by the following query Q.

$$Q := {}^{\mathtt{Q}}\Lambda^{\{\mathtt{Dmonth,Dyear,Pcat}\}}_{\{\mathtt{SUM(gross)\ as\ gross}\}}({}^{\mathtt{Q_1}}\Pi^{\{\mathtt{Dmonth,Dyear,Pcat,gross}\}}_{\{\mathtt{Pid{=}STprod}\}}(\mathtt{P},\mathtt{Q_{1.1}}))$$

Suppose that the database contains the following materialized view V, which stores the yearly cost and profit for sales broken down by product and store, giving the cumulative totals for each date (not just month's end).

$$V := {}^{\mathtt{V}}\Lambda^{\{\mathtt{Ddate,Deom,Pcat,Pid,Pname,STstore}\}}_{\{\mathtt{SUM(profit)\ as\ profit,\ SUM(cost)\ as\ cost}\}}({}^{\mathtt{V_1}}\Pi^{\{\mathtt{Ddate,Deom,Pcat,Pid,Pname,STstore,cost,profit}\}}_{\{\mathtt{Ddate{\geq}STdate,Dyear{=}year(STdate)}\}}(\mathtt{D},\mathtt{V_{1.1}}))$$

$$V_{1.1} := {}^{\mathtt{V_{1.1}}}\Lambda^{\{\mathtt{Pcat,Pid,Pname,STdate,STstore}\}}_{\{\mathtt{SUM(Pcost)\ as\ cost,\ SUM(STprice{-}Pcost)\ as\ profit}\}}({}^{\mathtt{V_{1.1.1}}}\Pi^{\{\mathtt{Pcat,Pcost,Pid,Pname,STdate,STprice,STstore}\}}_{\{\mathtt{Pid{=}STprod}\}}(\mathtt{P},\mathtt{ST}))$$

Assume that the navigation decisions made by NEWMATCH attempt to match subqueries in Q with subviews in V of corresponding depths (i.e. the algorithm chooses match style 2 every time). We will examine the results of each of these matches as they are generated in bottom-up order.

NEWMATCH($Q_{1.1.1}$, $V_{1.1.1}$):
Before normalization, the algorithm calculates the rewrite

$$Q_{1.1.1} := \Pi^{\{\mathtt{Dmonth,Dyear,STprice,STprod}\}}_{\{\mathtt{STdate{\leq}Ddate,year(STdate){=}Dyear,Deom{=}Y}\}}(\Phi^{\{\mathtt{Pcat,Pcost,Pid,Pname,STdate,STprice}\};\ \{\mathtt{STprod}\}}_{\{\mathtt{Pid{=}STprod}\}}[\mathtt{P}](\mathtt{V_{1.1.1}}))$$

which is then normalized via Rule PPU1 to return

$$Q_{1.1.1} := \Phi^{\{\mathtt{Dmonth,Dyear,Pcat,Pcost,Pid,Pname,STprice}\};\ \{\mathtt{STprod}\}}_{\{\mathtt{Pid{=}STprod}\}}[\mathtt{P}](\Pi^{\{\mathtt{Dmonth,Dyear,Pcat,Pcost,Pid,Pname,STprice}\}}_{\{\mathtt{STdate{\leq}Ddate,year(STdate){=}Dyear,Deom{=}Y}\}}(\mathtt{V_{1.1.1}},\mathtt{D}))$$

NEWMATCH($Q_{1.1}$, $V_{1.1}$):
Starting with

$$Q_{1.1} := {}^{\mathtt{Q_{1.1}}}\Lambda^{\{\mathtt{Dmonth,Dyear,STprod}\}}_{\{\mathtt{SUM(STprice)\ as\ gross}\}}(\mathtt{Q_{1.1.1}})$$

where $Q_{1.1.1}$ is defined as rewritten above, the algorithm applies the sequence of Rules PPU2, PD2 to push the prerequisites and compensation from $Q_{1.1.1}$ above the operator $Q_{1.1}$. This gives[8]

$$Q_{1.1} := \Phi^{\{\mathtt{Dmonth,Dyear,Pcat,Pid,Pname}\};\ \{\mathtt{STprod}\}}_{\{\mathtt{Pid{=}STprod}\}}[\mathtt{P}](\Lambda^{\{\mathtt{Dmonth,Dyear,Pcat,Pid,Pname}\}}_{\{\mathtt{SUM(gross)\ as\ gross}\}}(\mathtt{X}))$$
$$X := \Pi^{\{\mathtt{Dmonth,Dyear,Pcat,Pid,Pname,gross}\}}_{\{\mathtt{STdate{\leq}Ddate,year(STdate){=}Dyear,Deom{=}Y}\}}(\mathtt{D},{}^{\mathtt{Q'_{1.1}}}\Lambda^{\{\mathtt{Pcat,Pid,Pname,STdate}\}}_{\{\mathtt{SUM(STprice)\ as\ gross}\}}(\mathtt{V_{1.1.1}}))$$

The algorithm then reasons that the grouping list in $Q'_{1.1}$ is a subset of the grouping list in $V_{1.1}$, and that the expression SUM(STprice) can be rewritten as SUM(SUM(STprice−Pcost)+SUM(Pcost)). Therefore, $Q'_{1.1}$ is replaced with an aggregation operator over $V_{1.1}$, giving

$$Q_{1.1} := \Phi^{\{\mathtt{Dmonth,Dyear,Pcat,Pid,Pname}\};\ \{\mathtt{STprod}\}}_{\{\mathtt{Pid{=}STprod}\}}[\mathtt{P}](\Lambda^{\{\mathtt{Dmonth,Dyear,Pcat,Pid,Pname}\}}_{\{\mathtt{SUM(gross)\ as\ gross}\}}(\mathtt{X}))$$
$$X := \Pi^{\{\mathtt{Dmonth,Dyear,Pcat,Pid,Pname,gross}\}}_{\{\mathtt{STdate{\leq}Ddate,year(STdate){=}Dyear,Deom{=}Y}\}}(\mathtt{D},\Lambda^{\{\mathtt{Pcat,Pid,Pname,STdate}\}}_{\{\mathtt{SUM(profit{+}cost)\ as\ gross}\}}(\mathtt{V_{1.1}}))$$

which is already in normal form (i.e. prerequisites are above compensations, and none of the extra relations in the prerequisite can cancel out a rejoin in the compensation).

[8]Note that throughout this example we will arbitrarily break up a query body into sub-pieces X, X', etc., simply for presentation. The definitions of X, X', etc., should be considered temporary, and will be redefined in each query expression. As well, we will temporarily prepend labels to certain operators in order to allow us to refer to then in the body of the text. These labels have no bearing on the definition of the query, and the same operator may be given a completely new label in a later query expression.

NEWMATCH(Q_1, V_1):
Starting with

$$Q_1 := {}^{\texttt{Q}_1}\Pi^{\{\texttt{Dmonth,Dyear,Pcat,gross}\}}_{\{\texttt{Pid=STprod}\}}(\texttt{P}, \texttt{Q}_{\texttt{1.1}})$$

where $Q_{1.1}$ is defined as rewritten above, the algorithm applies the Rule PR1 to resolve the top prerequisite operator in $Q_{1.1}$ against the root of Q_1, which removes the predicate Pid=STprod and the relation P. The remaining compensation operators in $Q_{1.1}$ can then be pulled above Q'_1 (which now is only a projection) using Rules PPU1 and PPU4, which gives us

$$\begin{aligned} Q_1 &:= \Lambda^{\{\texttt{Dmonth,Dyear,Pcat,Pid,Pname}\}}_{\{\texttt{SUM(gross) as gross}\}}(\Pi^{\{\texttt{Dmonth,Dyear,Pcat,Pid,Pname,gross}\}}_{\{\texttt{STdate}\leq\texttt{Ddate,year(STdate)=Dyear,Deom=Y}\}}(\texttt{D}, \texttt{X})) \\ X &:= \Lambda^{\{\texttt{Pcat,Pid,Pname,STdate}\}}_{\{\texttt{SUM(profit+cost) as gross}\}}({}^{\texttt{Q}'_1}\Pi^{\{\texttt{Pcat,Pid,Pname,cost,profit}\}}_{\{\}}(\texttt{V}_{\texttt{1.1}}))\end{aligned}$$

Now that all child prerequisites and compensations are moved out of Q'_1, the algorithm creates a new prerequisite operator to hold any unmatched predicates or relations from V_1, and then replaces Q'_1 with V_1, adding the attribute STdate to the missing set of the new prerequisite since it is not available from V_1.

$$\begin{aligned} Q_1 &:= {}^{\texttt{C}_1}\Lambda^{\{\texttt{Dmonth,Dyear,Pcat,Pid,Pname}\}}_{\{\texttt{SUM(gross) as gross}\}}({}^{\texttt{C}_2}\Pi^{\{\texttt{Dmonth,Dyear,Pcat,Pid,Pname,gross}\}}_{\{\texttt{STdate}\leq\texttt{Ddate,year(STdate)=Dyear,Deom=Y}\}}(\texttt{D}, \texttt{X})) \\ X &:= {}^{\texttt{C}_3}\Lambda^{\{\texttt{Pcat,Pid,Pname,STdate}\}}_{\{\texttt{SUM(profit+cost) as gross}\}}({}^{\texttt{P}_1}\Phi^{\{\texttt{Ddate,Deom,Pcat,Pid,Pname,cost,profit}\};\ \{\texttt{STdate}\}}_{\{\texttt{Ddate}\geq\texttt{STdate,year(STdate)=Dyear}\}}[\texttt{D}](\texttt{V}_1))\end{aligned}$$

The rewrite is now complete, except that it is not in the required normal form because the prerequisite operator P_1 is below the three compensation operators. We can apply Rule PPU3 to move P_1 above C_3, which generates a new aggregation prerequisite P_2. P_1 can then be resolved against C_2 using Rule PR1 (cancelling out the D relation and all of the predicates in P_1), and then moved past it using Rule PPU1, giving us

$$\begin{aligned} Q_1 &:= {}^{\texttt{C}_1}\Lambda^{\{\texttt{Dmonth,Dyear,Pcat,Pid,Pname}\}}_{\{\texttt{SUM(gross) as gross}\}}({}^{\texttt{P}_1}\Phi^{\{\texttt{Ddate,Deom,Pcat,Pid,Pname,gross}\};\ \{\texttt{Dmonth,Dyear}\}}_{\{\}}[\ ](\texttt{X})) \\ X &:= {}^{\texttt{C}_2}\Pi^{\{\texttt{Ddate,Deom,Pcat,Pid,Pname,gross}\}}_{\{\texttt{Deom=Y}\}}({}^{\texttt{P}_2}\Omega^{\{\texttt{Ddate,Deom,Pcat,Pid,Pname}\};\ \{\texttt{STdate}\}}_{\{\texttt{SUM(profit+cost) as gross}\};\ \{\}}(\texttt{X}')) \\ X' &:= {}^{\texttt{C}_3}\Lambda^{\{\texttt{Ddate,Deom,Pcat,Pid,Pname}\}}_{\{\texttt{SUM(profit+cost) as gross}\}}(\texttt{V}_1)\end{aligned}$$

The prerequisite P_1 cannot be moved above C_1 because P_1 has attributes in its missing list that are needed by C_1. Therefore, we must first use Rule RCI1 to introduce a Π operator below P_1 that re-joins the relation D in order to obtain the missing attribute Dmonth and Dyear (which is possible because Ddate is available in P_1, and Σ implies that $\{\texttt{Ddate}\} \to Id_{\texttt{D}}$). Then, P_1 becomes trivial and can be removed with Rule PI, and the newly created Π operator can be merged into C_2 using Rule VF1, leaving

$$\begin{aligned} Q_1 &:= {}^{\texttt{C}_1}\Lambda^{\{\texttt{Dmonth,Dyear,Pcat,Pid,Pname}\}}_{\{\texttt{SUM(gross) as gross}\}}({}^{\texttt{C}_2}\Pi^{\{\texttt{Dmonth,Dyear,Pcat,Pid,Pname,gross}\}}_{\{\texttt{Deom=Y,Ddate=Ddate}\}}(\texttt{D}, \texttt{X})) \\ X &:= {}^{\texttt{P}_2}\Omega^{\{\texttt{Ddate,Deom,Pcat,Pid,Pname}\};\ \{\texttt{STdate}\}}_{\{\texttt{SUM(profit+cost) as gross}\};\ \{\}}({}^{\texttt{C}_3}\Lambda^{\{\texttt{Ddate,Deom,Pcat,Pid,Pname}\}}_{\{\texttt{SUM(profit+cost) as gross}\}}(\texttt{V}_1))\end{aligned}$$

(Note that the above query expression does not translate to a syntactically pure SQL statement, as the references to attributes Ddate and Deom in C_2 are ambiguous. However, the desired semantics should be fairly obvious, and this problem would be resolved if we added features for attribute renaming to our query language.)

The next step in the normalization is to attempt to pull up prerequisite operator P_2. Applying Rule PPU4, we obtain

$$\begin{aligned} Q_1 &:= {}^{\texttt{C}_1}\Lambda^{\{\texttt{Dmonth,Dyear,Pcat,Pid,Pname}\}}_{\{\texttt{SUM(gross) as gross}\}}({}^{\texttt{P}_2}\Omega^{\{\texttt{Dmonth,Dyear,Pcat,Pid,Pname}\};\ \{\texttt{STdate}\}}_{\{\texttt{SUM(profit+cost) as gross}\};\ \{\}}(\texttt{X})) \\ X &:= {}^{\texttt{C}_2}\Pi^{\{\texttt{Dmonth,Dyear,Pcat,Pid,Pname,gross}\}}_{\{\texttt{Deom=Y,Ddate=Ddate}\}}(\texttt{D}, {}^{\texttt{C}_3}\Lambda^{\{\texttt{Ddate,Deom,Pcat,Pid,Pname}\}}_{\{\texttt{SUM(profit+cost) as gross}\}}(\texttt{V}_1))\end{aligned}$$

to which we can apply Rule PR3a to resolve the aggregation prerequisite P_2 against the aggregation operator C_1 (by reasoning that SUM(SUM(profit+cost)) is a cleanly composable nesting of aggregation functions). Our final, normalized rewrite for Q_1 is

$$Q_1 :=^{\mathtt{C_1}}\Lambda^{\{\mathtt{Dmonth,Dyear,Pcat,Pid,Pname}\}}_{\{\mathtt{SUM(gross)\ as\ gross}\}}({}^{\mathtt{C_2}}\Pi^{\{\mathtt{Dmonth,Dyear,Pcat,Pid,Pname,gross}\}}_{\{\mathtt{Deom=Y,Ddate=Ddate}\}}(\mathtt{D},\mathtt{X}))$$
$$X :=^{\mathtt{C_3}}\Lambda^{\{\mathtt{Ddate,Deom,Pcat,Pid,Pname}\}}_{\{\mathtt{SUM(profit+cost)\ as\ gross}\}}(\mathtt{V_1})$$

NEWMATCH(Q, V):
Starting with

$$Q :=^{\mathtt{Q}}\Lambda^{\{\mathtt{Dmonth,Dyear,Pcat}\}}_{\{\mathtt{SUM(gross)\ as\ gross}\}}(\mathtt{Q_1})$$

where Q_1 is defined as rewritten above, the algorithm attempts to move the compensation operators above operator Q. Operator C_1 can be merged into its parent using Rule VF2. Then, operator C_2 can be pushed up using Rule PD2 (which requires a new aggregation operator to be added above C_2 because the aggregation of the original Q is not invariant to relation D).

$$Q := \Lambda^{\{\mathtt{Dmonth,Dyear,Pcat}\}}_{\{\mathtt{SUM(gross)\ as\ gross}\}}({}^{\mathtt{C_2}}\Pi^{\{\mathtt{Dmonth,Dyear,Pcat,gross}\}}_{\{\mathtt{Deom=Y,Ddate=Ddate}\}}(\mathtt{D},\mathtt{X}))$$
$$X :=^{\mathtt{Q'}}\Lambda^{\{\mathtt{Ddate,Deom,Pcat}\}}_{\{\mathtt{SUM(gross)\ as\ gross}\}}({}^{\mathtt{C_3}}\Lambda^{\{\mathtt{Ddate,Deom,Pcat,Pid,Pname}\}}_{\{\mathtt{SUM(profit+cost)\ as\ gross}\}}(\mathtt{V_1}))$$

Operator C_3 then can be merged into its parent using Rule VF2 (which reasons that the resulting aggregation expression SUM(profit+cost) is decomposable and equivalent to the original composition SUM(SUM(profit+cost))). Now that all of the compensation operators have been moved above operator Q', the algorithm reasons that the grouping set of Q' is a subset of the grouping set of V, and so Q' can be replaced by a re-aggregation of view V (which requires reasoning that SUM(profit+cost) is distributive and equivalent to SUM(SUM(profit)+SUM(cost))). This generates the query expression

$$Q := \Lambda^{\{\mathtt{Dmonth,Dyear,Pcat}\}}_{\{\mathtt{SUM(gross)\ as\ gross}\}}(\Pi^{\{\mathtt{Dmonth,Dyear,Pcat,gross}\}}_{\{\mathtt{Deom=Y,Ddate=Ddate}\}}(\mathtt{D},\mathtt{X}))$$
$$X := \Lambda^{\{\mathtt{Ddate,Deom,Pcat}\}}_{\{\mathtt{SUM(profit+cost)\ as\ gross}\}}(\mathtt{V})$$

which is already in normal form. Note the compensation operators could be simplified to remove the inner aggregation (by Rule PU2), but this is not necessary for correctness. Therefore, the algorithm returns the above rewriting of Q to use view V.

It is not enough that the final rewriting be in the normal form (i.e. prerequisites operators above compensation operators). Instead, we have the added requirement that the final rewriting may not contain *any* prerequisite operators. The rewriting generated by the topmost matching in Example 8 already met this requirement; however, it is possible that the rewrite returned by the topmost match still contains a prerequisite operator. At this point we would need to find a sequence of simplifying transformations that would remove the prerequisite in order for the final rewrite to be valid. For example, the returned re-writing might be rooted by a Φ operator that lists an extra relation R; in order for the rewrite to be usable, either the extra R needs to resolve with a re-joined R below it in the compensation (a simplification which should have already been performed by the function SIMPLIFY-AND-NORMALIZE()), or R needs to be logically deleted from the Φ operator using Rule PCR, after which the Φ needs to be removed using Rule PI.

7.3 Implementation Issues

The focus of this research has been on improving the recall of previously published algorithms for rewriting complex aggregation queries using complex aggregation views. In presenting our extensions, we have not addressed what impact our modifications have on efficiency of the algorithm. There are three features of our modified algorithm `NEWMATCH` that negatively impact performance compared to the bottom-up approach described by Zaharioudakis et al.:

1. The cost of replacing templates with transform rules.
2. The cost of multiple equivalent paths through the algorithm.
3. The cost of generating rewrites containing unsatisfiable prerequisites.

The remainder of this section will address each of these costs in turn.

7.3.1 The Cost of Using Transform Rules

As described in Section 4, the use of simple templates to drive the matching is efficient, but limited in power, while driving the matching by iterative application of a set of transformation rules is powerful, but potentially expensive. We can reap the benefits of both approaches by creating templates for the simple cases, and using transform rules if none of the templates apply. Of course, this means the cost of trying the transformation rules is still incurred for every failed match. It should be noted, however, that our algorithm never applies tranformation rules blindly in a forward-chaining manner. Instead, every use of transformation rules is goal-oriented, and so we expect that an efficient backward-chaining process can be devised to locate a successful sequence of transformations, if one exists.

7.3.2 The Cost of Multiple Paths

An unfortunate side-effect of adding prerequisite operators to the matching algorithm is that they allow the matching styles 1 and 3 to interact such that a choice 1 followed by a later choice 3 (or vice-versa) can be equivalent to a single choice 2. The net effect is to create multiple paths through the algorithm that generate the same rewriting. The following example illustrates the problem.

Example 9 Assume that the following query Q and view V are phrased over the sales database schema from Example 3.

$$Q := {}^{\mathtt{Q}}\prod^{\{\mathtt{Pname,Sname,STprice}\}}_{\{\mathtt{Pid=STprod,Page>18}\}}(\mathtt{P}, {}^{\mathtt{Q_1}}\prod^{\{\mathtt{Sname,STprice,STprod}\}}_{\{\mathtt{Sid=STstore,Scat=3}\}}(\mathtt{S},\mathtt{ST}))$$
$$V := {}^{\mathtt{V}}\prod^{\{\mathtt{Page,Pname,Scat,Sname,STprice}\}}_{\{\mathtt{Pid=STprod}\}}(\mathtt{P}, {}^{\mathtt{V_1}}\prod^{\{\mathtt{Scat,Sname,STprice,STprod}\}}_{\{\mathtt{Sid=STstore}\}}(\mathtt{S},\mathtt{ST}))$$

There are three different paths through the algorithm `NEWMATCH(`Q`,`V`)` that generate the identical rewriting.

Path 1:

```
Call NEWMATCH($Q$, $V$)
   Choose match style 2
   Call NEWMATCH($Q_1$, $V_1$)
      Choose match style 2
      Return $Q_1' := \prod^{\{\mathtt{Sname,STprice,STprod}\}}_{\{\mathtt{Scat=3}\}}(\mathtt{V_1})$
   Return $Q' := \prod^{\{\mathtt{Pname,Sname,STprice}\}}_{\{\mathtt{Page>18,Scat=3}\}}(\mathtt{V})$
```

Path 2:

```
Call NEWMATCH($Q$, $V$)
   Choose match style 3
   Call NEWMATCH($Q$, $V_1$)
      Choose match style 1
      Call NEWMATCH($Q_1$, $V_1$)
         Choose match style 2
         Return $Q_1' := \prod_{\{\texttt{Scat=3}\}}^{\{\texttt{Sname,STprice,STprod}\}}(\texttt{V}_1)$
      Return $Q' := \prod_{\{\texttt{Pid=STprod,Page>18,Scat=3}\}}^{\{\texttt{Pname,Sname,STprice}\}}(\texttt{P},\texttt{V}_1)$
   Call NEWMATCH($Q'$, $V$)
      Choose match style 2
      Return $Q'' := \prod_{\{\texttt{Page>18,Scat=3}\}}^{\{\texttt{Pname,Sname,STprice}\}}(\texttt{V})$
   Return $Q''$
```

Path 3:

```
Call NEWMATCH($Q$, $V$)
   Choose match style 1
   Call NEWMATCH($Q_1$, $V$)
      Choose match style 3
      Call NEWMATCH($Q_1$, $V_1$)
         Choose match style 2
         Return $Q_1' := \prod_{\{\texttt{Scat=3}\}}^{\{\texttt{Sname,STprice,STprod}\}}(\texttt{V}_1)$
      Return $Q_1'' := \Phi_{\{\texttt{Pid=STprod}\}}^{\{\texttt{Page,Pname,Sname,STprice}\};\ \{\texttt{STprod}\}}[\texttt{P}](\prod_{\{\texttt{Scat=3}\}}^{\{\texttt{Page,Pname,Sname,STprice}\}}(\texttt{V}))$
   Return $Q' := \prod_{\{\texttt{Page>18,Scat=3}\}}^{\{\texttt{Pname,Sname,STprice}\}}(\texttt{V})$
```

Obviously, the different paths would be avoided if we first flattened Q and V using Rule `VF1`. However, the same behaviour can be observed among queries and views containing aggregation operators that cannot be flattened.

In Example 9, Path 1 performs a one-to-one match between the blocks of V and the blocks of Q; hence, this path mimics the behaviour of the original bottom-up algorithm. Path 2 matches the root of Q against both V and V_1 but still uses only compensation operators; therefore, this path was enabled by our adding of match style 3 to algorithm `MATCH`. Path 3 matches Q_1 against both V and V_1, using prerequisite operators to do to; thus, this path was enabled by our adding of prerequisite operators to algorithm `NEWMATCH`.

The main disadvantage to having multiple equivalent paths through a top-down algorithm is that an exhaustive search of the space will perform redundant work. In the `NEWMATCH` algorithm as given, the number of equivalent paths (and hence the redundant work) grows exponentially in the number of operators in the query and view definitions. However, there is a high amount of overlap in the recursive invocations made by different paths. Thus, if a memo structure is used to avoid redundancy due to re-invocation, the redundancy due to differing choices of matching style is at most quadratic in the number of operators in the query and view definitions. (The same quadratic factor would be introduced if prerequisites and matching style 3 were added to the equivalent purely bottom-up algorithm.) In practice, it is unlikely that a quadratic scale-up would actually be observed, even when attempting to find all possible rewrites. However, it is left for future work to design heuristics that attempt to decide when the different match types should be tried or can be avoided.

7.3.3 The Cost of Unsatisfiable Prerequisites

By design, prerequisite operators are intended to avoid early pruning of usable views in an attempt to improve the recall of the matching process. However, unconstrainted use of prerequisites can destroy efficiency by delaying pruning of unusable views. For example, a view V containing a non-logically-deletable relation R that occurs nowhere in query Q should be pruned as soon as R is encountered; but, by creating a Φ prerequisite that lists R as an extra relation, algorithm NEWMATCH could conceivably proceed much farther through the matching process, only to discover at the end that the prerequisite cannot be resolved.

In an ideal world, our algorithm would use an oracle to tell it which prerequisites could be resolved later in the matching process, and it would only create those prerequisites, pruning any views that required a prerequisite that could not be later resolved. This would improve the recall of the original algorithm without introducing any false positives at intermediate stages. Unfortunately, constructing a perfect oracle requires global knowledge of the query and view definitions, destroying the efficiency of the bottom-up process that makes local decisions. However, if we replace the oracle with an approximation that does not require global knowledge, we can still hope to improve recall without introducing too many false positives. Different choices in designing oracle-approximations lead to different levels of trade-off between false positives and negatives in the matching process.

One possible oracle-approximation can be constructed by an approach that we will call *Ancestor Inspection.* Given an invocation of `NEWMATCH(`Q`,`V`)` in which Q is actually a subquery within a larger query tree, Ancestor Inspection examines only the operators occurring as ancestors to Q within the outer query tree (including the grouping sets, aggregation expressions, predicate sets, and joined base relations within these operators) in order to decide whether the given prerequisite is resolvable. It should now be clear why we have written `NEWMATCH` to navigate the query tree from the top-down rather than bottom-up—the information required by Ancestor Inspection is not locally available when traversing the query definition bottom-up. In contrast, a top-down traversal can carry forward memory of Q's ancestor operators using space proportional to the depth of Q in the larger query tree (or constant space, if we approximate this memory using hashing), allowing the decision to be made based upon locally-available information.

Ancestor Inspection is inherently conservative, because it rejects prerequisites that could be resolved against compensation operators originating out of an ancestor's sibling (such as the scenario illustrated in Example 7). Initially converting the query to a normal form can help to improve the recall of Ancestor Inspection by maximizing the opportunity to resolve against operators along the ancestor path.

Definition 7.1 (Ancestor Normal Form (ANF)) *A query expression Q composed of standard relational operators is said to be in* Ancestor Normal Form *iff*

1. *The transformations `VF1` (flattening direction), `PD1`, and `PD3` cannot be applied anywhere within Q.*

2. *Each predicate is located at its* divergent point.

Definition 7.2 (Divergent Point) *A predicate p_1 is located at its* divergent point *if it is pushed down as far as possible so that there does not exist a predicate p_2 among the ancestor operators of p_1 such that*

1. *p_2 is an equality predicate;*

2. *$attributes(p_1) \cap attributes(p_2) \neq \emptyset$; and*

3. *p_2 is at its divergent point.*

The intuition behind the first condition of Ancestor Normal Form is that in order to maximize the opportunity to resolve a prerequisite containing an extra relation, all relations should be pulled up as high as possible. Rules VF1, PD1, and PD3 all cause relations to move upward in the query tree, thus increasing the size of the descendant subtree for which they would be visible for Ancestor Inspection. The intuition behind the second condition of ANF is that pushing a predicate p_1 down past its divergent point removes it from the ancestor path of the subquery to which p_2 joins it, which could cause Ancestor Inspection to reject resolvable prerequisites within that subquery.

Converting the initial query to ANF gives the added benefit that the function SIMPLIFY-AND-NORMALIZE described in Section 7.1 does not need to search to resolve prerequisites and compensations originating out of sibling subqueries.

Example 10 The query Q from Example 7 can be converted to the following query Q' which is in Ancestor Normal Form.

$$Q' :=^{\mathtt{Q'}} \Pi^{\mathtt{Sid,Sname,ecount,sales}}_{\mathtt{Estore=Sid,Sid=STstore,Estore>100}}(\mathtt{S}, \mathtt{Q'_1}, \mathtt{Q'_2})$$

$$Q'_1 :=^{\mathtt{Q'_1}} \Lambda^{\mathtt{STstore}}_{\mathtt{SUM(STprice)\ as\ sales}}(\mathtt{ST})$$

$$Q'_2 :=^{\mathtt{Q'_2}} \Lambda^{\mathtt{Estore}}_{\mathtt{COUNT(Eid)\ as\ ecount}}(\mathtt{E})$$

Observe that the divergent point for the predicate ``Estore>100'' is within the operator $Q'$—not $Q'_2$—due to its overlap with the predicate ``Estore=Sid'' which can not be pushed down any further.

Consider the view V also from Example 7. Ancestor Inspection would disallow the rewriting of Q to use V described in Example 7 because there is no reason to believe that prerequisite operator generated by matching Q_1 with V_1 has any chance of being resolved, based upon the predicates and relations along Q_1's ancestor path (i.e. neither the relation ``S'' nor the predicate ``STstore>100'' are found in either Q_1 or Q). In contrast, Ancestor Inspection permits the rewriting of Q' to use V because the prerequisite formed when matching Q'_1 with V_1 is potentially resolvable by the predicates and relations occurring in the ancestor operator Q'.

Through the use of heuristics such as Ancestor Inspection in conjunction with normal forms such as ANF, we believe that prerequisite operators can be used to improve matching recall without seriously hampering the ability to prune unusable views early.

8 Conclusions

Using prerequisite operators is a promising strategy for improving the recall of known algorithms that perform view matching for multi-block aggregation queries in a bottom-up fashion. Although there are serious performance drawbacks associated with unrestrained introduction of prerequisite operators, a judicious governing of prequisite introduction (through the use of heuristics such as Ancestor Investigation in conjunction with normal forms such as ANF) should lead to increased matching recall with a limited increase in algorithmic overhead.

In the future we plan to investigate more strategies for limiting the introduction of prerequisites to cases where they are likely to be satisfiable. We also plan on investigating completeness of the matching algorithm with respect to a set of known transformation rules, as well as the impact of the afore-mentioned governing strategies on this completeness.

References

[1] R. G. Bello, K. Dias, A. Downing, J. J. F. Jr., J. L. Finnerty, W. D. Norcott, H. Sun, A. Witkowski, and M. Ziauddin. Materialized views in Oracle. In A. Gupta, O. Shmueli,

and J. Widom, editors, *Proceedings of 24th International Conference on Very Large Data Bases (VLDB'98)*, pages 659–664. Morgan Kaufmann, 1998.

[2] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26(1):65–74, 1997.

[3] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *Proceedings of 20th International Conference on Very Large Data Bases (VLDB'94)*, pages 354–366. Morgan Kaufmann, 1994.

[4] S. Chaudhuri and K. Shim. Optimizing queries with aggregate views. In P. M. G. Apers, M. Bouzeghoub, and G. Gardarin, editors, *Advances in Database Technology - Proceedings of the 5th International Conference on Extending Database Technology (EDBT'96)*, volume 1057 of *Lecture Notes in Computer Science*, pages 167–182. Springer, 1996.

[5] R. J. Cochrane, G. Lapis, T. Y. Leung, M. H. Pirahesh, Y. Sun, and M. Zaharioudakis. Cube indices for relational database management systems. U.S. Patent 6,560,594, May 2003.

[6] S. Cohen, W. Nutt, and Y. Sagiv. Equivalences among aggregate queries with negation. In *Proceedings of the 20th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2001)*, pages 215–226. ACM Press, 2001.

[7] S. Cohen, W. Nutt, and Y. Sagiv. Containment of aggregate queries. In *Proceedings of the 9th International Conference on Database Theory (ICDT 2003)*, volume 2572 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2003.

[8] S. Cohen, W. Nutt, and A. Serebrenik. Rewriting aggregate queries using views. In *Proceedings of the 18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 1999)*, pages 155–166. ACM Press, 1999.

[9] D. DeHaan, D. Toman, and G. Weddell. Rewriting aggregate quereis using description logic. In *Proceedings of the 2003 International Workshop on Description Logics (DL2003)*, pages 103–112, September 2003. Available at http://CEUR-WS.org/Vol-81/.

[10] C. Galindo-Legaria and M. M. Joshi. Cost based materialized view selection for query optimization. U.S. Patent 6,510,422, January 2003.

[11] J. Goldstein and P.-A. Larson. Optimizing queries using materialized views: a practical, scalable solution. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 331–342. ACM Press, 2001.

[12] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible query processing in Starburst. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 377–388. ACM Press, 1989.

[13] D. S. Johnson and A. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. *Journal of Computer and System Sciences*, 28(1):167–189, 1984.

[14] J. C. Mitchell. The implication problem for functional and inclusion dependencies. *Inf. Control*, 56(3):154–173, 1983.

[15] W. Nutt, Y. Sagiv, and S. Shurin. Deciding equivalences among aggregate queries. In *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 1998)*, pages 214–223. ACM Press, 1998.

[16] W. P. Yan and P.-A. Larson. Performing group-by before join. In *Proceedings of the 10th International Conference on Data Engineering (ICDE'94)*, pages 89–100. IEEE Computer Society, 1994.

[17] W. P. Yan and P.-A. Larson. Eager aggregation and lazy aggregation. In *Proceedings of 21st International Conference on Very Large Data Bases (VLDB'95)*, pages 345–357. Morgan Kaufmann, 1995.

[18] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering complex SQL queries using automatic summary tables. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 105–116. ACM Press, 2000.