Constructing GPSJ View Graphs

Michael O. Akinde
Department of Computer Science
Aalborg University
Fredrik Bajers Vej 7 E-1
DK–9220 Aalborg Øst, Denmark
strategy@cs.auc.dk

Michael H. Böhlen
Department of Computer Science
Aalborg University
Fredrik Bajers Vej 7 E-1
DK–9220 Aalborg Øst, Denmark
boehlen@cs.auc.dk

Abstract

A data warehouse collects and maintains integrated information from heterogeneous data sources for OLAP and decision support. An important task in data warehouse design is the selection of views to materialize, in order to minimize the response time and maintenance cost of generalized project-select-join (GPSJ) queries.

We discuss how to *construct* GPSJ view graphs. GPSJ view graphs are directed acyclic graphs, used to compactly encode and represent different possible ways of evaluating a set of GPSJ queries. Our view graph construction algorithm, GPSJVIEWGRAPHBUILDER, incrementally constructs GPSJ view graphs based on a set of merge rules. We provide a set of merging rules to construct GPSJ view graphs in the presence of duplicate sensitive and insensitive aggregates. The merging algorithm used in GPSJVIEWGRAPHBUILDER ensures that each node is correctly added to the view graph, and employs the merge rules to ensure that relationships between nodes from different queries are incorporated into the view graph.

Proceedings of the International Workshop on Design and Management of Data Warehouses (DMDW'99)
Heidelberg, Germany, 14. - 15.6. 1999
(S. Gatziu, M. Jeusfeld, M. Staudt, Y. Vassiliou, eds.)
http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-19/

1 Introduction

A data warehouse is a repository of integrated information from multiple, independent data sources available for querying and analysis. As data warehouses contain integrated information, often spanning long periods of time, they tend to be orders of magnitude larger than conventional operational databases; ranging from hundreds of gigabytes to terabytes in size. The workload is typically query-intensive, with many complex queries that may access millions of records and perform many joins and aggregates.

Three costs must be balanced during physical database design for warehouses: (1) the cost of answering queries, (2) the cost of maintaining the warehouse, and (3) the cost of secondary storage. The cost of (1) can be reduced by materializing (precomputing) frequently asked queries as materialized views in the data warehouse, but this increases the maintenance costs of the warehouse. The problem of selecting an appropriate set of views and indexes to materialize in a data warehouse is referred to as the *view-selection* [Gup97] or *data warehouse configuration* problem [TS97].

For the purpose of our discussion, we use the following terminology (precise definitions follow later). A *GPSJ query* is a generalized project-select-join query, i.e., a project-select-join query extended with aggregation, grouping, and group selection. This class of queries is the single most important one used in data warehousing [Kim96]. A *GPSJ query graph* is a directed acyclic graph. It represents a specific strategy to evaluate a GPSJ query. A *GPSJ expression DAG* compactly encodes different possibilities to evaluate a GPSJ query. It combines multiple GPSJ query graphs into a single graph. Finally, a *GPSJ view graph* encodes multiple evaluation strategies for different queries. Put differently, a GPSJ view graph integrates several GPSJ expression DAGs.

In this paper, we concentrate on the *construction* of GPSJ view graphs. The *integrator* takes the set of GPSJ

expression DAGs as input, and merges them into a GPSJ view graph according to a set of merging rules. The integrator and its embedding into the data warehouse configuration tool are illustrated in Figure 1. Given a set of queries for the data warehouse and metadata (such as query frequencies, schema information, etc.), the tool returns a data warehouse configuration.

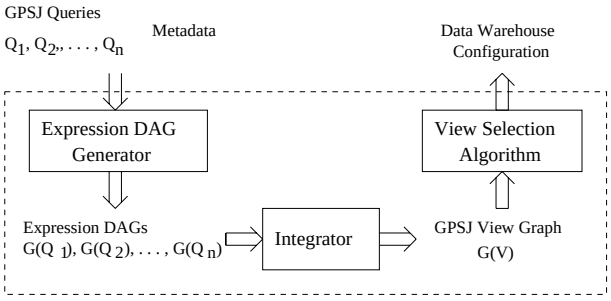

Figure 1: Data Warehouse Configuration Tool

Example 1.1 Consider the table:

```
accounts(a_personid, a_name, a_balance,
         a_type, a_date)
```

and the SQL queries:

```
A1 SELECT   a_name,max(a_balance)
   FROM     accounts
   GROUP BY a_name

A2 SELECT   a_personid,a_name,
            sum(a_balance)
   FROM     accounts
   GROUP BY a_name
```

Figure 2 gives two simple expression DAGs, $G(A1)$ and $G(A2)$, for these two queries. Equivalence nodes, representing views which can be materialized in the data warehouse, are denoted in the figures using ovals and a label. Operation nodes are denoted using rectangles. The integrator takes these expression DAGs as input, and merges them to derive the GPSJ view graph $G(X)$ displayed on the left. Note that during the merging process, additional operation nodes and edges may be derived, such as the node and edges connecting A' to A_1.

A major part of the integrator is the GPSJVIEWGRAPHBUILDER algorithm. The algorithm starts with a (possibly empty) GPSJ view graph and incrementally builds up the "final" graph by integrating GPSJ expression DAGs into it. The behavior of the algorithm is controlled by a set of merging rules, which are also part of the integrator. This flexibility allows us to extend the GPSJ view graph framework to consider more general view graphs and various graph input types, simply by adding or modifying the set of merge rules used in the GPSJVIEWGRAPHBUILDER algorithm.

Specifically, given a set of data warehouse queries, $Q_1, Q_2, \ldots Q_n$, the GPSJ view graph $G(X)$ is incrementally constructed from the set of associated GPSJ expression DAGs, $G(Q_1), G(Q_2), \ldots, G(Q_n)$ by the algorithm using a set of merging rules. We give a set of basic rules to integrate GPSJ expression DAGs into GPSJ view graphs. To the best of our knowledge, the rules and algorithms for merging such graphs in the presence of aggregation and grouping have not previously been considered in the literature.

In order to ensure an optimal solution to the view-selection problem, it is necessary to generate the entire GPSJ view graph. However, the complexity (space and time) of algorithms for solving the view-selection problem optimally is exponential in the number of nodes in the expression graph [TS97, Gup97, GM99]. Although we do not directly represent the conventional (duplicate-preserving) projection in the GPSJ view graphs of this framework, as they can be discarded when they appear as interior nodes in the graph [RSS96, YKL97] and can be expressed using a generalized projection when appearing as the outermost nodes [GHQ95], the complete GPSJ view graph in the presence of aggregation and grouping alone can still be huge. Therefore, we provide a number of rules which can be used to reduce the size of the view graph.

1.1 Related Work

The problem of constructing view graphs such as those described in this paper is most often considered in relation to view-selection algorithms or physical database design. Roussopolous [Rou82] presents an algorithm for generating LAP schemas, which closely resemble our view graphs, however, his algorithm does not consider aggregate functions and grouping.

Other papers on view-selection employing view-graph like structures [RSS96, TS97, GM99] either do not consider the construction of these structures, or consider only select-join views without aggregation.

Gupta [Gup97] suggests that the AND-OR view graph be constructed using the expression AND-OR DAGs of the queries. These expression AND-OR DAGs are to contain only those views which will be considered (useful) for the view selection algorithm. However, it is unclear how to determine these "useful" views in the presence of aggregation, and therefore how to construct the AND-OR view graph. This is the problem considered in this paper.

1.2 Paper Outline

The paper is structured as follows. Section 2 describes the aggregation framework and notation used in this paper, and gives rules for recomputing aggregates using their disjoint sets. Section 3 defines the GPSJ view graphs, and gives the

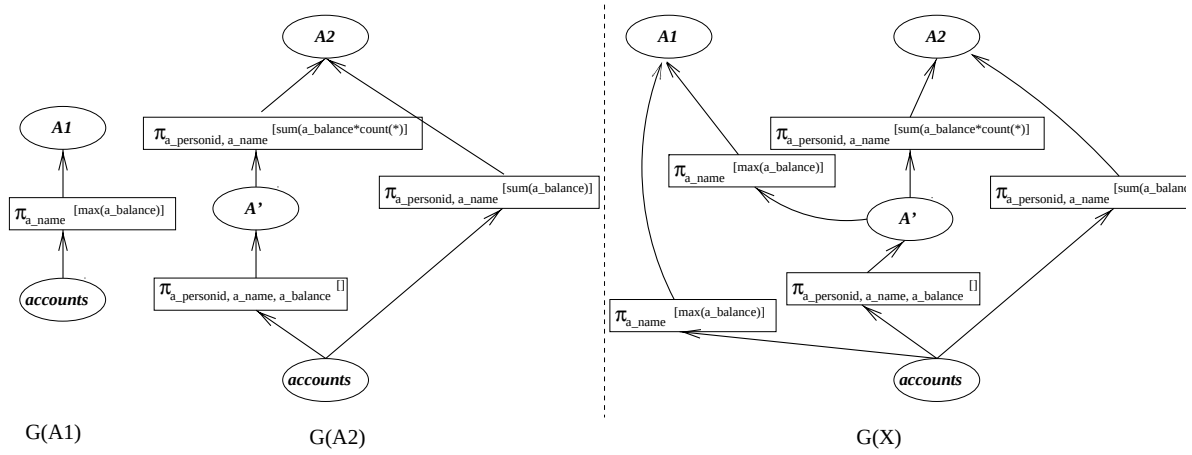

Figure 2: Combining two GPSJ Expression DAGs into a GPSJ View Graph

graphical notation used to illustrate the graphs. Section 4 gives rules and the algorithm for merging expression DAGs for GPSJ queries. In Section 5 we give some rules for reducing the size of the GPSJ view graph. Section 6 concludes the paper and points to future research directions.

2 Aggregation Framework

We use the generalized projection (GP) operator [GHQ95], $\pi_G[F(A)]$, to represent aggregation. Generalized projection is an extension of duplicate eliminating projection, where G denotes the set of group-by attributes and F denotes a set of aggregate functions $F = f_1, f_2, \ldots, f_n$ over attributes in the attribute set A.

In this paper, we consider only distributive aggregate functions, i.e., aggregate functions that can be computed by partitioning their inputs into disjoint sets. The SQL aggregate functions `count`, `sum`, `min`, and `max`, are all distributive. The algebraic aggregate function `avg` can be expressed in this framework using `sum`/`count`.

Aggregate functions can be divided into the duplicate sensitive aggregates (referred to as DSAs), such as `count` and `sum`, and the duplicate insensitive aggregates (referred to as non-DSAs), such as `max` and `min`. These characteristics are of importance when computing an aggregate function from its disjoint sets. In general, non-DSAs can always be computed from views which contain the same aggregate, or the attribute of the aggregate; e.g., $\pi_A[max(B)](\pi_{A,B}R) \equiv \pi_A[max(B)](R)$. In order to do a similar transformation with DSAs, we need additional information about the number of duplicates. This information can be acquired using a `count`. We refer to the process of computing aggregates from their group-by attributes using a `count` as duplicate compensation.

Table 1 gives the rules for computing aggregates from the partial results of previously computed aggregates or the group-by attributes. The prerequisite attributes are those group-by attribute and/or aggregate functions required in a view for the computation of the aggregate in the first column. For example, we can compute $Q = \pi[sum(A)]R$ given the view $V = \pi_A[\texttt{count}(*)\ \texttt{as}\ cnt](R)$ as $Q = \pi[\texttt{sum}(A * cnt)](V)$.

Aggregate	Prerequisite Attributes	Computed Aggregate
`count(`$*$`)`	`count(`$*$`)` as cnt	`sum(`cnt`)`
`count(`A`)`	`count(`A`)` as $cntA$	`sum(`$cntA$`)`
`count(`A`)`	A,`count(`$*$`)` as cnt	`sum(`cnt`)`
`sum(`A`)`	`sum(`A`)` as $sumA$	`sum(`$sumA$`)`
`sum(`A`)`	A, `count(`$*$`)` as cnt	`sum(`$A * cnt$`)`
`max(`A`)`	`max(`A`)` as $maxA$	`max(`$maxA$`)`
`max(`A`)`	A	`max(`A`)`
`min(`A`)`	`min(`A`)` as $minA$	`min(`$minA$`)`
`min(`A`)`	A	`min(`A`)`

Table 1: Computing Aggregates from partial results and/or group-by attributes

An example of the application of such rules is the deriving of $A1$ or $A2$ from the node A' in Figure 2.

3 GPSJ View Graphs

A GPSJ query is PSJ query enhanced with grouping and aggregation. More precisely, a GPSJ query is any query which can be written in GP normal form [GHQ95] (i.e., a selection, σ_1, over a generalized projection, π, over a selection, σ_2, over a set of joins, $\mathcal{X}$,: $\sigma_1\pi\sigma_2\mathcal{X}$). A large class of queries can be expressed as GPSJ queries, in particular all `SELECT-FROM-WHERE-GROUP BY-HAVING` queries can be reduced to this form if the attributes/aggregate functions in the `GROUP BY` and `HAVING` clauses appear in the `SELECT` clause, no aggregate functions use the `DISTINCT` keyword, and the `WHERE` clauses are conjunctive. Algorithms for solving the view-selection prob-

lem [Gup97, TS97, GM99], usually model the problem as some form of graph structure representing multiple queries. We use GPSJ view graphs for this purpose.

A *query graph* for a query Q is a graph, where each leaf node corresponds to a base table used to define Q, and each non-leaf node is an operator with associated children. The algebraic expression computed at the root node is equivalent to Q. Query graphs are used in query optimizers to determine the cost of a particular way of evaluating a query. We refer to the leaves and root nodes of the query graph as "equivalence" nodes and the non-leaf nodes as "operation" nodes.

Expression DAGs are used to compactly represent the space of the equivalent query graphs of a single query as a directed acyclic graph. An expression DAG is a bipartite directed acyclic graph with equivalence nodes and operation nodes. An equivalence node (with the possible exception of leaf nodes) has edges to one or more operation nodes. An operation node consists of an operator, edges to either one or more predecessors that are equivalence nodes, and an edge to the derived equivalence node. We denote an equivalence node by the algebraic expression it computes. Its predecessor operation nodes correspond to various query graphs that yield a result that is algebraically equivalent to the label of the equivalence node. The leaves of an expression DAG are equivalence nodes corresponding to base tables. Expression DAGs are used in rule-based optimizers.

The GPSJ view graph is a multi-query expression DAG, i.e., it represents expression DAGs of several queries in a single DAG. Each equivalence node in the view graph corresponds to a view, which can be materialized in a data warehouse. Like the expression DAG, the leaf equivalence nodes of a GPSJ view graph correspond to the base tables of the data warehouse. We define the equivalence nodes of GPSJ view graphs as follows:

Definition 3.1 (Equivalence Nodes of View Graphs) *Let $\mathcal{R}$ be the set of base tables in the data warehouse. We define the set of equivalence nodes $\mathcal{V}$ of a GPSJ view graph recursively as:*

1. *If $R_i \in \mathcal{R}$, then $R_i \in \mathcal{V}$.*
2. *If $R_i, R_j \in \mathcal{V}$, then:*
   - *(a) $\sigma[C](R_i) \in \mathcal{V}$, where $\sigma[C](R_i)$ selects the subset of the tuples of R_i that satisfies the condition C.*
   - *(b) $\pi_G[F](R_i) \in \mathcal{V}$, where $\pi_G[A]$ is a generalized projection on R_i. G is a subset of the attributes in R_i and F are a set of aggregate functions over R_i.*
   - *(c) $\bowtie[C](R_i, R_j) \in \mathcal{V}$, where $\bowtie[C](R_i, R_j)$ is a join between R_i and R_j on the join condition C. The Cartesian product of two views is represented using a join on an empty condition ($C = \emptyset$).*

In GPSJ expression DAGs and GPSJ view graphs the three basic operations are denoted using the graphical notation of Figure 3.

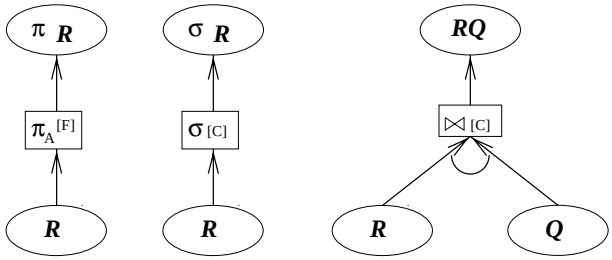

Figure 3: (a) Generalized Projection, (b) Selection, and (c) Join in GPSJ View Graphs

Definition 3.2 (GPSJ View Graphs) *A directed acyclic graph $G(X)$ having the base tables as the leaves is called a GPSJ view graph for the queries (or views) $Q_1, Q_2, \ldots, Q_n$ if for each query Q_i, there is a subgraph $G_i(Q_i)$ in $G(X)$ that is an expression DAG of Q_i. Each equivalence node v in the GPSJ view graph is annotated with the query frequency f_v (frequency of queries on v), update frequency g_v (frequency of update on v), and the size s_v of the view if materialized.*

Update frequence and size of the views in the GPSJ view graph can be re-calculated after the construction of the GPSJ view graph. Therefore, the only parameter of interest during the *construction* of the GPSJ view graph is the query frequency f_v. We will not consider the update frequency or size in the rest of this paper. Also, to avoid cluttering up the figures, we will not annotate the graphs with query frequencies in the examples.

An important characteristic of the GPSJ view graph is its similarity to AND-OR view graphs. This means that standard view selection algorithms [Gup97, GM99] can, with little or no modification, be used on GPSJ view graphs.

4 Constructing GPSJ View Graphs

The GPSJ view graph of a set of queries $Q_1, Q_2, \ldots, Q_n$ is constructed by merging the expression DAGs $\mathcal{G}(\mathcal{Q}) = G(Q_1), G(Q_2), \ldots, G(Q_n)$ for each of these queries.

Example 4.1 Consider the table `accounts` of Example 1.1 and the SQL query :

```
SELECT   a_name, max(a_balance)
FROM     accounts
GROUP BY a_name
```

Figure 4 gives three equivalent query graphs for this query, and the expression DAG derived from these query graphs.

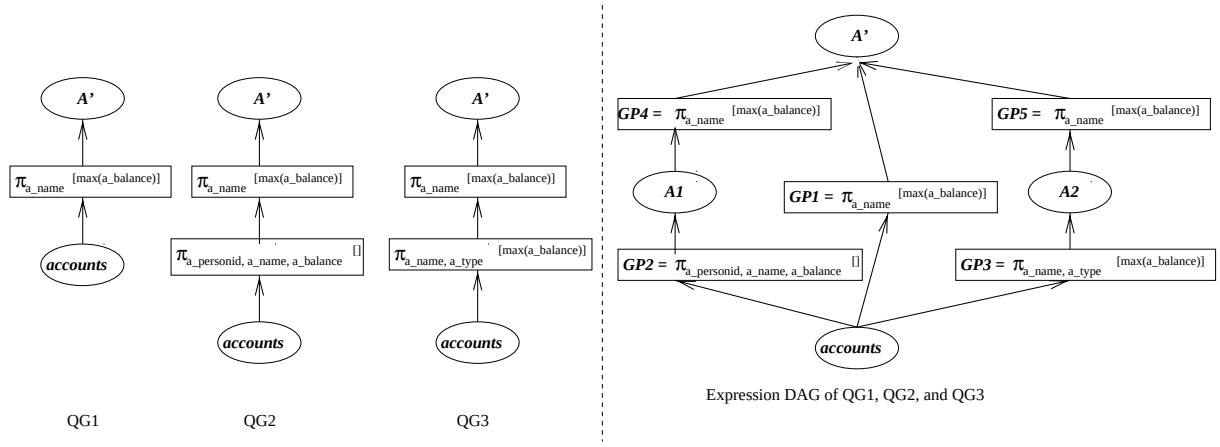

Figure 4: Equivalent query graphs and the resulting expression DAG

For the purposes of this paper, we assume that the complete expression DAG has been generated. This issue has no effect on the construction algorithm itself, as the inputs of this algorithm is simply a set of graphs, however complete expression DAGs are required to construct complete GPSJ view graphs. The complete expression DAG of aggregate queries are typically very large, e.g., the space complexity of the number of equivalence nodes in a simple aggregate query (i.e., one constructed using a single projection over a base table πR) is $O(2^{n-m})$, where m is the number of attributes used in the view, and n is the number of attributes in the base table.

For example, the complete expression DAG of the query in Example 4.1 has $2^{5-2} = 8$ different possible groupings, namely all combination of attributes that include `a_name` and `a_balance`. Using these 8 groupings we end up with over 51 possible distinct query graphs. In addition to these, we could also construct query graphs with `max(a_balance)` instead of grouping on `a_balance`, resulting in more than a hundred possible query graphs for this query.

For the purposes of algorithms such as those presented in [Gup97, GM99], we can not simply choose an "optimal" query graph, as we might then miss important equivalence nodes which could be shared by other queries. However, without knowledge about the other queries being considered, it is impossible to know which views, and thus, which nodes of the view graph will be useful for the purposes of the view-selection algorithm, without sacrificing the optimality of the solution. In Section 5, we will consider rules for reducing the size of GPSJ view graphs. Given information about the set of queries being considered, this can be done without sacrificing the performance guarantee of the view selection algorithms being used.

The incremental merging of each expression DAG $G(Q_i)$ of $\mathcal{G}(\mathcal{Q})$ into the view graph, is controlled by the set of merging rules which we discuss below. The merge rules ensure that potential structural relationships between equivalence nodes in different DAGs of $\mathcal{G}(\mathcal{Q})$ are correctly incorporated in the GPSJ view graph.

1. Let the initial GPSJ View Graph $G(X)$ contain equivalence nodes corresponding to the base relations used in the queries $Q_1, Q_2, \ldots, Q_n$.
2. Annotate each equivalence node in $\mathcal{G}(\mathcal{Q})$ with f_v.
3. For each $G(Q_i) \in \mathcal{G}(\mathcal{Q})$: MERGEALGORITHM($G(X)$,$G(Q_i)$) (cf. Figure 10)

Figure 5: GPSJVIEWGRAPHBUILDER

4.1 Merge Rules

The rules and merge algorithm are specified so as to ensure that it is not necessary to iterate in the graph to discover whether two queries can be computed from each other using a single operation. Assuming that the full expression DAG of a query has been materialized, three rules are sufficient to ensure that all derivations between nodes in the graphs of two different graphs will be derived. Note that the set of views handled by the merge algorithm (i.e., GPSJ views) can easily be extended by the introduction of additional merge rules for other operators (e.g., outer join, union, etc.).

We use standard inference rules on the form $\psi, \phi \vdash \varphi$ to state that, given ψ and ϕ, we can infer φ.

Rule 4.1 (Derivation from a Selection Node)

$$V_1 = \sigma[C_1](V), V_2 = \sigma[C_2](V) \vdash V_1 = \sigma[C_1](V_2)$$

iff the selection condition C_1 restricts the same attributes as C_2, and the condition C_1 is more restrictive than C_2.

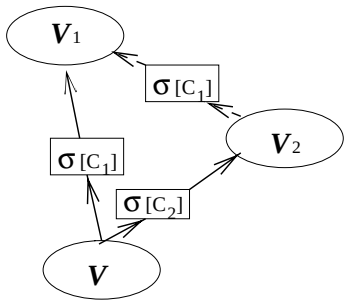

Figure 6: Illustration of Rule 4.1

Rule 4.2 (Derivation from a GP Node)

$V_1 = \pi_{G_1}[F_1](V), V_2 = \pi_{G_2}[F_2](V) \vdash V_1 = \pi_{G_1}[F'](V_2)$

iff:

1. $G_1 \subseteq G_2$,
2. *for each DSA* $f_i(a_i) \in F_1$ *there exists a corresponding DSA* $f_i(a_i) \in F_2$, *or a* $count(*) \in F_2$ *and* $a_i \in G_2$, *and*
3. *for each non-DSA* $f_i(a_i) \in F_1$ *there exists a corresponding non-DSA* $f_i(a_i) \in F_2$ *or* $a_i \in G_2$.

The set of aggregate functions F' *is derived from* F_1 *and* F_2 *using the rules for recomputing distributive aggregate functions from their component parts (see Table 1).*

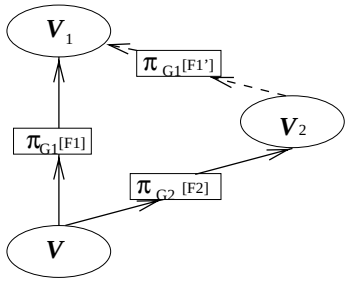

Figure 7: Illustration of Rule 4.2

Rule 4.3 (Derivation from a Join Node)

$$V_1 = \bowtie[C_1](V_i, V_j), V_2 = \bowtie[C_2](V_i, V_j) \vdash V_1 = \sigma[C_1](V_2)$$

iff the join condition C_1 *restricts the same attributes as* C_2, *and the condition* C_1 *is more restrictive than* C_2.

Rule 4.3 can also be used to handle the view V derived from a Cartesian product. This is done by treating the Cartesian product as a join on the empty condition, i.e., any join condition C will always be more restrictive than the conditions of the Cartesian product.

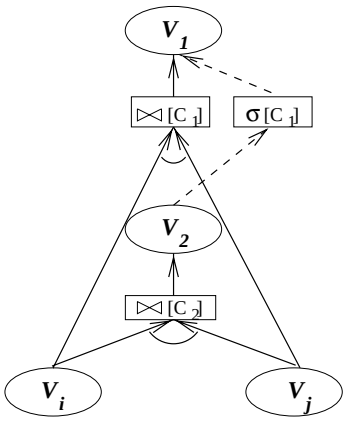

Figure 8: Illustration of Rule 4.3

4.2 The Merging Algorithm

Before we present the algorithm for merging, we shall briefly describe the notation of the algorithm. We let $G(X)$ and $G(Q)$ represent the GPSJ view graph and the expression DAG of the query Q, respectively. We formally describe an expression DAG or view graph, following the description in Section 3.

Definition 4.1 (Graph Definition) *A DAG or view graph* $G(X)$ *is a tuple* $\langle V_X, O_X \rangle$, *where* V_X *is the set of equivalence nodes, and* O_X *is the set of operation nodes.*

Each equivalence node $v \in V_X$ *is a triple* $\langle v_L, O_p, O_d \rangle$, *where* v_L *is the label of the equivalence node.* O_p *and* O_d *are a set of labels, where* O_p *is the set of predecessor operations and* O_d *is the set of derivative operations.*

Each operation node $o \in O_X$ *is a triple* $\langle o_L, E_p, E_d \rangle$, *where* o_L *is the label of the operation node.* E_p *and* E_d *are a set of labels, where* E_p *denotes one or more predecessor equivalence nodes, and* E_d *is the derived equivalence node. We refer to* E_p *and* E_d *as the predecessor and derivative expressions.*

Example 4.2 Consider the expression DAG in Figure 4. Following Definition 4.1, we can define the expression DAG $G(X)$ as follows:

$$\begin{aligned}
G(X) =& \langle V_X, O_X \rangle \\
V_X =& \{\langle accounts, \{\}, \{GP1, GP2, GP3\}\rangle, \\
& \langle A1, \{GP2\}, \{GP4\}\rangle, \\
& \langle A2, \{GP3\}, \{GP5\}\rangle, \\
& \langle A', \{GP1, GP4, GP5\}, \{\}\rangle\} \\
O_X =& \{\langle GP1, \{accounts\}, \{A'\}\rangle, \\
& \langle GP2, \{accounts\}, \{A1\}\rangle, \\
& \langle GP3, \{accounts\}, \{A2\}\rangle, \\
& \langle GP4, \{A1\}, \{A'\}\rangle, \\
& \langle GP5, \{A2\}, \{A'\}\rangle\}
\end{aligned}$$

We define the graph as a set of equivalence nodes and operation nodes (rather than as a set of nodes and edges) as these two kinds of nodes are treated separately in the algorithm. Recall our claim in Section 3 regarding the similarity of GPSJ view graphs to AND-OR view graphs. To transform an expression DAG or view graph defined as above into an AND-OR DAG, we simply transform the operation nodes into edges; each operation node corresponds to an AND arc in the AND-OR DAG framework.

We refer to the operation nodes connecting an equivalence node to its derivative nodes as derivative operation nodes, and the operation nodes connecting it to its deriving equivalence nodes as predecessor operation nodes.

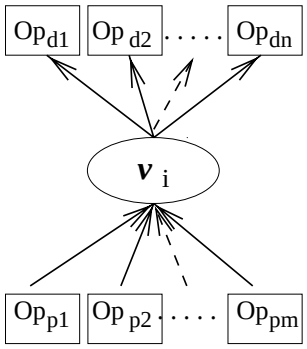

Figure 9: Predecessor and derivative operation nodes of the equivalence node v_i

Definition 4.2 (Predecessor & Derivative Operations) *Given a DAG as in Figure 9, we define the predecessor operations* PreOps *and derivative operations* DerOps *of the equivalence node v_i as:*

$$PreOps(v_i) = \{Op_{p1}, Op_{p2}, \ldots, Op_{pm}\}$$
$$DerOps(v_i) = \{Op_{d1}, Op_{d2}, \ldots, Op_{dn}\}$$

To test whether a view v_i is a child (i.e., directly derived) of v_j, we test $PreOps(v_i) \cap DerOps(v_j) \neq \emptyset$.

In order to simplify the presentation of the algorithm, we assume that the list of the views V_Q of the expression DAG $G(Q)$ is an ordered set of views $v_1, v_2, \ldots, v_n$ such that no view v_i is derived from a view v_{i+j}, where i and j are positive numbers. Such an ordering is easily imposed using a topological sort. The MERGEALGORITHM is given in Figure 10. The explanations follow below.

The intuition behind the merging algorithm is as follows. We consider each view v_i of the query graph $G(Q)$ in connection with the views of $G(X)$, considering the leaves/base tables of $G(Q)$ first. The time complexity of MERGEALGORITHM is $O(|V_X| * (|V_X| + |V_Q|))$, assuming that no views in $G(Q)$ match with those in $G(X)$, which is a worst case.

If the views v_i and v_j are identical, we merge the two views into a single equivalence node in $G(X)$, using the **Merge** function, which is defined as follows:

Input
The GPSJ View Graph $G(X)$
The Expression DAG $G(Q)$
Output
The modified GPSJ View Graph $G(X)'$
Method
Let $G(X) = \langle V_X, O_X \rangle$
Let $G(Q) = \langle V_Q, O_Q \rangle$
If $V_Q = \emptyset$ then
  Return $G(X)$.
For each $v_i \in V_Q$:
  For each $v_j \in V_X$:
    %% Step 1 - Check for Equality
    If $v_i = v_j$ then do
      Merge(v_i, v_j)
      Let $f_{v_j} = f_{v_j} + f_{v_i}$.
    Else
      %% Step 2 - Check Ancestry
      If v_i is a child of v_j and $v_i \notin V_X$
      then
        Add(v_i,$G(X)$).
      Else
        %% Step 3 - Attempt to Derive
        If v_i is derivable from v_j using
        Rules 4.1-4.3 or (v_j is derivable
        from v_i using Rules 4.1-4.3 and
        v_j is not a child of v_i) then do
          If $v_i \notin V_X$ then
            Add(v_i,$G(X)$).
          Add the deriving operation node
          to v_i and v_j.
        EndIf
  End For
End For
Return $G(X)$.

Figure 10: MERGEALGORITHM($G(X), G(Q)$)

Merge($\mathbf{v_1}, \mathbf{v_2}$)
Let $G(X) = \langle V_X, O_X \rangle$
Let $\mathcal{E}$ be the set of expressions
  in the operation nodes
  denoted by the labels of
  $DerOps(v_1)$.
If $v_1 \in \mathcal{E}$ then:
  $\mathcal{E} = (\mathcal{E} \setminus \{v_1\}) \cup \{v_2\}$
$DerOps(v_2) = DerOps(v_2) \cup DerOps(v_1)$
Let O_{v1} be the set of operation
  nodes denoted by the labels
  of $DerOps(v_1)$.
$O_X = O_X \cup O_{v_1}$

Example 4.3 Consider the view graph $G(X) = \langle V_X, O_X \rangle$ and the DAG $G(Q) = \langle V_Q, O_Q \rangle$, where:

$$V_X = \{\langle acc1, \{\}, \{GP1\}\rangle, \langle A1, \{GP1\}, \{\}\rangle\}$$
$$O_X = \{\langle GP1, \{acc1\}, \{A1\}\rangle\}$$
$$V_Q = \{\langle acc2, \{\}, \{GP2\}\rangle, \langle A2, \{GP2\}, \{\}\rangle\}$$
$$O_Q = \{\langle GP2, \{acc2\}, \{A2\}\rangle\}$$

We assume $acc1$ and $acc2$ are equivalent and/or represent the same view and perform **Merge**($acc2$, $acc1$). This re-

sults in the following configurations:

$$DerOps(acc2) = \{GP2\}$$
$$\mathcal{E} = \{acc2, A2\}$$

We then replace $acc2$ with $acc1$, to get $\mathcal{E} = \{acc1, A2\}$. Finally, we add $DerOps(acc2)$ to $DerOps(acc1)$ and add the operation node $GP2$ to O_X to obtain the following configuration of $G(X)$:

$$V_X = \{\langle acc1, \{\}, \{GP1, GP2\}\rangle, \langle A1, \{GP1\}, \{\}\rangle\}$$
$$O_X = \{\langle GP1, \{acc1\}, \{A1\}\rangle, \langle GP2, \{acc1\}, \{A2\}\rangle\}$$

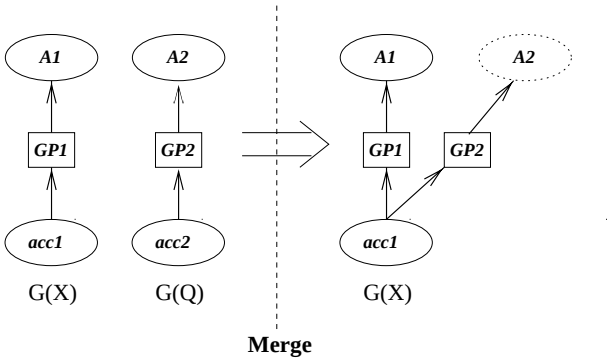

Figure 11: Illustration of Example 4.3

To add the operations (or set of operations) of an equivalence node v_1 to v_2, we first update the expressions of the derivative operation nodes referencing v_1 to reference v_2. We then add the derivative operations of v_1 to v_2, and add the operation nodes to the graph. The merge function ensures that any child v_k of v_i will also be detected as a child of v_j. Note that this implies that equivalence nodes in $G(X)$ can actually have operations connecting them to equivalence nodes in $G(Q)$ during the processing of the merge algorithm (as shown in Example 4.3). Due to the ordering imposed on V_Q however, we are always ensured that any predecessor equivalence nodes will be merged into the view graph first. Finally, we update the query frequency f_v of the merged view in $G(X)$.

At the lowest level of the query graph, views correspond to base relations. In this case, recognizing identical views is a simple matter of name matching. For the levels above, the recognition is done by checking whether the derivation of a pair of views is equivalent. The problem of testing for equivalence of GPSJ queries is considered in [GHQ95, SDJL96, NSS98, RSSS98]. The problem can be simplified by normalizing the expressions used to identify the equivalence nodes when constructing the expression DAGs.

If the view v_i is a child of some v_j, and v_i or an identical view does not already exist in $G(X)$, then we add v_i to the GPSJ view graph $G(X)$, using the **Add** function:

$\mathbf{Add}(\mathbf{v}, \mathbf{G(X)})$
Let $G(X) = \langle V_X, O_X\rangle$, $v = \langle v_L, O_p, O_d\rangle$
Let O_v be the operation nodes
  denoted by the set of labels
  O_d.
$V_X = V_X \cup v$
$O_X = O_X \cup O_v$

Because of the ordering imposed on $G(Q)$, we only need to add the derivative operation nodes O_d of v, as the algorithm ensures that the predecessor nodes will have been added previously.

Finally, we apply Rules 4.1 to 4.3 to check whether either of the views v_i or v_j can be derived from the other. The test to check whether v_j is a child of v_i is required to avoid duplicate operation nodes being introduced in this step, since $G(X)$ is not an ordered list like $G(Q)$. Rules 4.1 to 4.3 provide us with information about the deriving operation node o', which can then be added to O_p and O_d of v_i and v_j as appropriate, to connect the two nodes. It is critical for the correctness of the view selection algorithm, that it has full information about the connectivity of the view graph.

Example 4.4 Consider the tables

```
accounts(a_personid,a_name,a_balance,
        a_type,a_date)
transaction(t_personid,t_change,t_date)
```

with the following three SQL queries:

```
Q1 SELECT   a_name,max(a_balance)
   FROM     accounts
   GROUP BY a_name

Q2 SELECT   a_name,count(t_personid)
   FROM     accounts,transaction
   WHERE    t_personid = a_personid
            AND t_change > 250
   GROUP BY a_name

Q3 SELECT   a_name,a_balance
   FROM     accounts,transaction
   WHERE    t_personid = a_personid
            AND t_change > 500
   GROUP BY a_name,a_balance
```

The expression DAGs are given in Figure 12. For simplicity of presentation, we give only very simple expression DAGs, each representing two possible query evaluation paths.

We initialize the GPSJ view graph $G(X)$ with the two base table nodes `accounts` and `transaction` (A and T in the figures). The initial merging of $Q1$ results in a "graph" similar to that of $Q1$ in Figure 12, except for the presence of the unconnected equivalence node T.

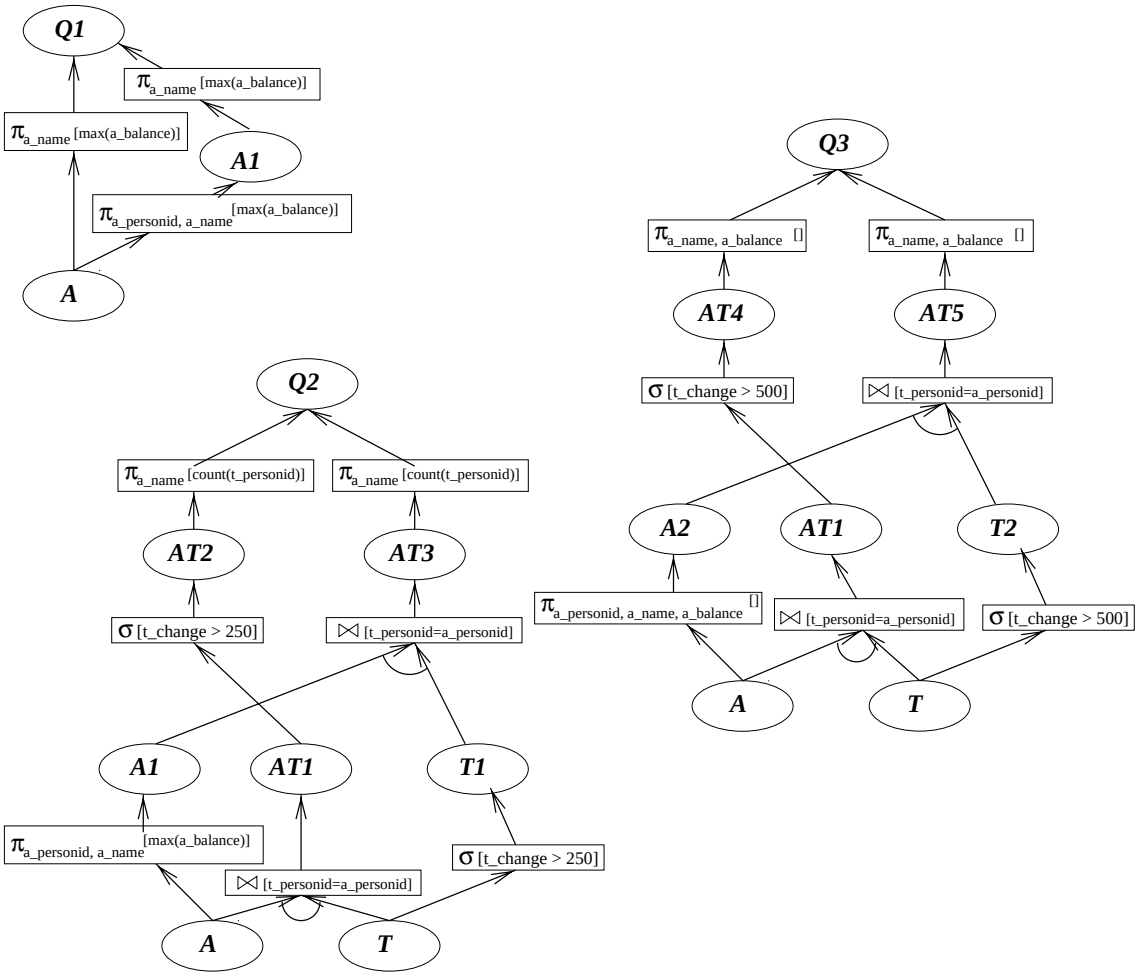

Figure 12: The Expression DAGs of the three queries Q1, Q2, and Q3

Figure 13 shows the GPSJ view graph after $Q2$ has been merged into the graph, and Figure 14 illustrates the complete view graph after $Q3$ has been merged into the graph. In Figure 14 additional operations have been added to the graph using the rules of Section 4.1 to derive $A1$ and $Q1$ from $A2$, $AT4$ from $AT2$, and $T2$ from $T1$.

In the view graph of Figure 14, each of the equivalence nodes represents a view which could be materialized in the data warehouse to answer one of the queries $Q1$, $Q2$, or $Q3$, while the paths in the graphs represent ways of computing the queries from these views. It is thus possible to apply a view selection algorithm to select and identify useful sets of views to materialize. For example, if we wished to select a set of views M, such that $M \cup \{Q1, Q2, Q3\}$ are self-maintainable (i.e., can be maintained on changes to the base tables without referencing these), we could materialize $A2$ and $T1$.

5 Reducing the GPSJ View Graph

The actual running time of view selection algorithms depends on the size and structure of the generated view graph. The full expression DAG of an aggregate query is typically huge (cf. the very simple query in Example 4.1); this results in a blow-up of the size of the view graph structure and longer running time or larger space requirements of the view selection algorithm. A solution to the performance problem of view selection algorithms is to prune the search space of the algorithm [TS97, YKL97]. This corresponds to reducing the size of GPSJ view graphs.

Obviously, a view selection algorithm cannot consider or select a view, if that view is not represented in the GPSJ view graph. Therefore, care must be taken to ensure that we do not remove equivalence nodes which might be considered by the view selection algorithm, if we wish to maintain a performance guarantee for the view selection, such as those presented in [Gup97, GM99].

Recall our assumption that the complete expression DAG is generated for each query, thus ensuring that the

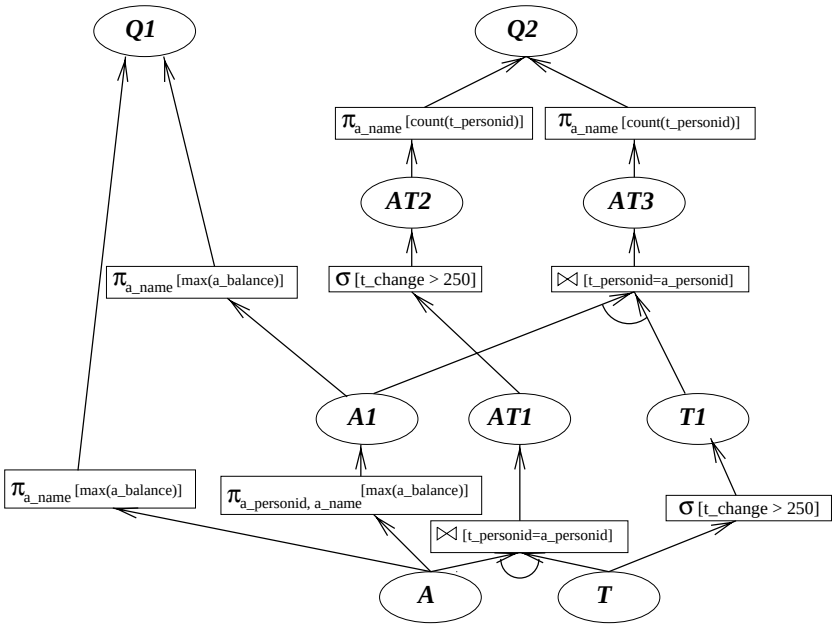

Figure 13: The view graph after merging Q1 and Q2

GPSJ view graph is also complete. However, given knowledge about the complete set of queries to be considered, it is possible to identify views which will never be considered by the view selection algorithm. Removing such views allows us to reduce the size and complexity of the GPSJ view graph structure.

We present two rules, which reduce the size of the GPSJ view graph without affecting subsequent view selection algorithms, i.e., these algorithms will still select the same set of views to materialize. This reduction, or pruning, of the graph structure can occur at several different stages of the configuration tool architecture (cf. Figure 1), either as a pre- or post-optimization of the Integrator component, or as part of the expression DAG generation.

Rule 5.1 (Pruning Group-by attributes) *Let A_i be the set of group-by attributes, attributes used in join and selection conditions, and attributes used in aggregate functions in Q_i. Then $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ includes all the attributes used in the queries $Q_1, Q_2, \ldots, Q_n$. Then the only views which are of interest for the view selection algorithm constructed using a GP are those with group-by A_{gb}, where $A_{gb} \subseteq \mathcal{A}$.*

Intuitively, we are only interested in those views which contain attributes used in the set of queries $Q_1, Q_2, \ldots, Q_n$. If we have a view $V_1 = \pi_{A_1,A_2}[F](V)$ in our expression DAG, where $A_1 \subseteq \mathcal{A}$ and $A_2 \not\subseteq \mathcal{A}$, then we can create a view $V_2 = \pi_{A_1}[F](V)$ which can be used to answer the same set of queries as V_1 and whose benefit (i.e., measure of the usefulness of the view) will always be greater than or equal to that of V_2. Since V_2 can replace V_1 with respect to the set of queries, it follows that V_1 is not a "useful" view for the view selection algorithm. Recall the $O(2^{n-m})$ space complexity for the number of equivalence nodes in the simple aggregate query expression DAG. Rule 5.1 reduces the number of group-by attributes considered in the expression DAG (the factor n), thus minimizing the exponent.

Before giving the second rule, we first define superfluous aggregates.

Definition 5.1 (Superfluous Aggregates) *An aggregate function $f_i(a_i)$ is superfluous, if it can be computed from other components of the same GP. Let $\pi_A[F]$ be a GP over a table. A non-DSA $f_i(a_i) \in F$ is superfluous if $a_i \in A$. A DSA $f_i(a_i) \in F$ is superfluous if $a_i \in A$ and $count(a_i) \in F$.*

Rule 5.2 (Eliminate Superfluous Aggregates) *Let A_{gb} be the set of group-by attributes and F be a set of aggregate function over the attributes A_{agg}, and $A_v = A_{gb} \cap A_{agg}$. If $V' = \pi_{A_{gb}}[F](V)$ is a view in the expression DAG and $A_v \neq \emptyset$, then we replace V' with a view $V'' = \pi_{A_{gb}}[F'](V)$. F' is identical to F, minus the superfluous aggregates. If a DSA f_i can be made superfluous by introducing a* `count` *function to F, we add the* `count` *and remove f_i.*

The correctness of Rule 5.2 follows from the principles of duplicate compensation (cf. Section 2). Since superfluous aggregates do not add additional value for the view selection algorithm (i.e., if a query can be computed from a view containing superfluous aggregates, then it can equally

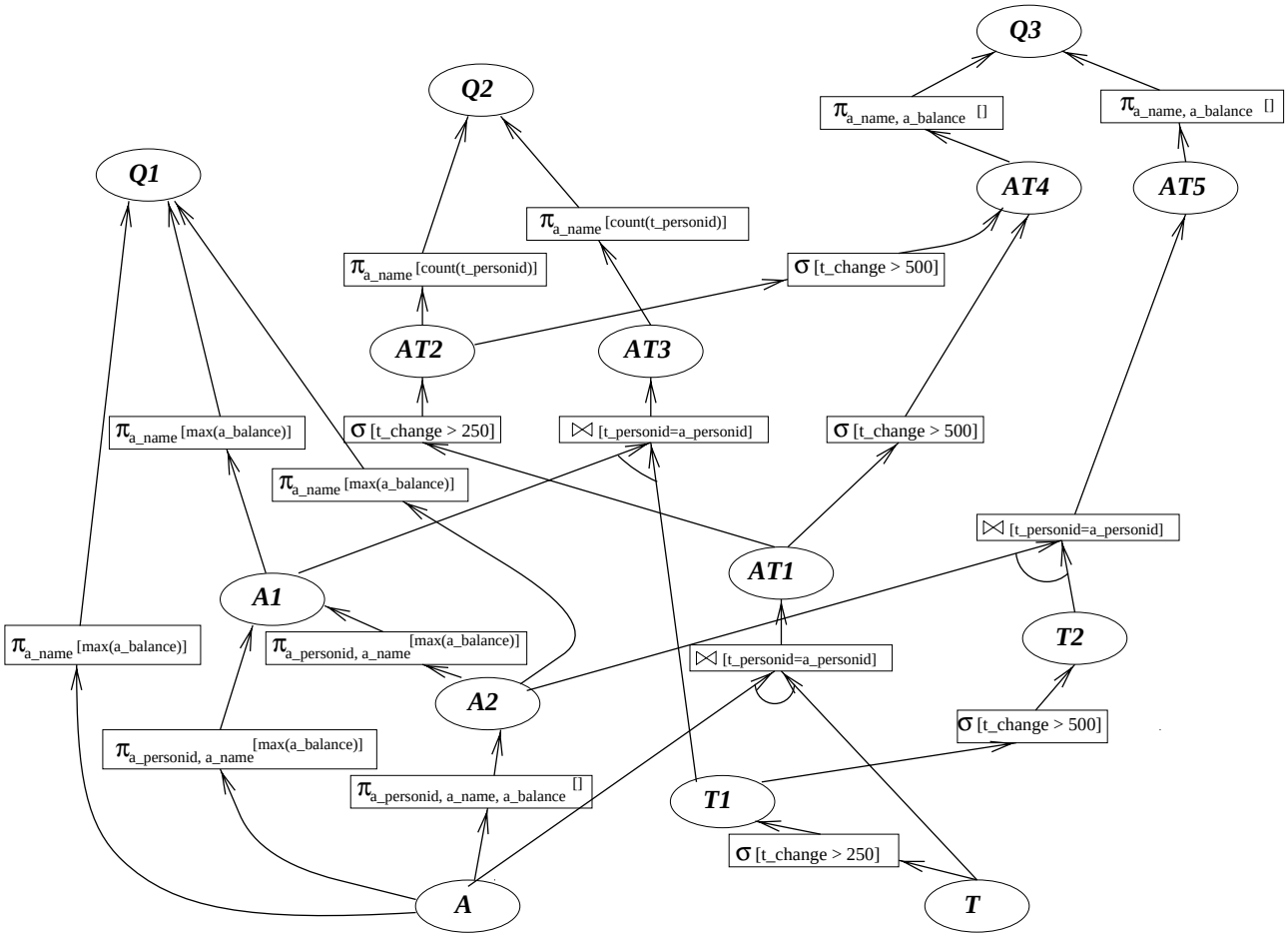

Figure 14: The view graph after merging Q1, Q2, and Q3

well be computed from one containing no superfluous aggregates), we eliminate such views to reduce the size of the view graph.

6 Conclusion

The selection of which views to materialize is one of the most important decisions in data warehouse design. The view selection (or data warehouse configuration) problem is to select a set of views to materialize in the data warehouse, so as to optimize the total query response time under some space or maintenance cost constraints. We discuss the view graph structures required by view selection algorithms such as those proposed in [Gup97, GM99]. We are not aware of any papers considering the construction of such graph structures in the presence of aggregation and grouping.

We describe an algorithm, GPSJVIEWGRAPH-BUILDER for the construction of GPSJ view graphs from the expression DAGs of GPSJ queries based on a set of merge rules. We define a set of rules for merging complete expression DAGs, and give a merge algorithm for carrying out the incremental merging of an expression DAG with the GPSJ view graph. We also give a number of rules for reducing the size of the view graph constructed, while still allowing us to keep the performance guarantee of the view selection algorithms used.

In our future work, we intend to investigate the following issues:

There is a lot of scope for reducing the size of the GPSJ view graph generated for the view selection algorithm. It would be interesting to attempt to further examine the effects of "non-optimal" pruning strategies on the view selection algorithms considered. Is it possible to tailor view selection algorithms to particular pruning strategies?

The GPSJ view graph framework can easily be extended with operations such as outer join, union, and other relational operators by extending the merge rules of the algorithm. This would allow us to consider a broader class of views.

References

[GHQ95] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-Query Processing in Data Warehousing Environments. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *Proceedings of the Twenty-first International Conference on Very large Databases*. Zurich, Switzerland, September 1995.

[GM99] Himanshu Gupta and Inderpal Singh Mumick. Selection of Views to Materialize Under a Maintenance Cost Constraint. To appear in the Proceedings of the ICDT'99, 1999.

[Gup97] H. Gupta. Selection of Views to Materialize in a Data Warehouse. In *Proceedings of the Sixth ICDT*, pages 98–112, 1997.

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

[NSS98] Werner Nutt, Yehoshua Sagiv, and Sara Shurin. Deciding equivalences among aggregate queries. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 214–223, Seattle, Washington, USA, June 1998. ACM Press.

[Rou82] N. Roussopolous. The Logical Access Path Schema of a Database. *IEEE Transactions in Software Engineering*, SE-8(6):563–573, November 1982.

[RSS96] Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 447–458, Montreal, Quebec, Canada, June 1996.

[RSSS98] K. A. Ross, D. Srivastava, P. J. Stuckey, and S. Sudarshan. Foundations of Aggregation Constraints. *Theoretical Computer Science*, 193:149–179, February 1998.

[SDJL96] Divesh Srivastava, Shaul Dar, H. V. Jagadish, and Alon Y. Levy. Answering Queries with Aggregation Using Views. In *Proceedings of the 22nd Annual International Conference on Very Large Data Bases*, Bombay, India, September 1996.

[TS97] Dimitri Theodoratos and Timos Sellis. Data Warehouse Configuration. In *Proccedings of the Twenty-third International Conference on Very Large Data Bases*, pages 126–135, Athens, Greece, August 1997.

[YKL97] J. Yang, K. Karlapalem, and Q. Li. Algorithms for Materialized View Design in Data Warehousing Environment. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *Proceedings of the Twenty-third International Conference on Very Large Databases*, August 1997.