Answering Complex SQL Queries Using Automatic Summary Tables

Markos Zaharioudakis, Roberta Cochrane, George Lapis, Hamid Pirahesh, Monica Urata

IBM Almaden Research Center
San Jose, CA 95120
{markos, bobbiec, lapis, pirahesh}@almaden.ibm.com, monicau@us.ibm.com

ABSTRACT

We investigate the problem of using materialized views to answer SQL queries. We focus on modern decision-support queries, which involve joins, arithmetic operations and other (possibly user-defined) functions, aggregation (often along multiple dimensions), and nested subqueries. Given the complexity of such queries, the vast amounts of data upon which they operate, and the requirement for interactive response times, the use of materialized views (MVs) of similar complexity is often mandatory for acceptable performance. We present a novel algorithm that is able to rewrite a user query so that it will access one or more of the available MVs instead of the base tables. The algorithm extends prior work by addressing the new sources of complexity mentioned above, that is, complex expressions, multidimensional aggregation, and nested subqueries. It does so by relying on a graphical representation of queries and a bottom-up, pair-wise matching of nodes from the query and MV graphs. This approach offers great modularity and extensibility, allowing for the rewriting of a large class of queries.

1. INTRODUCTION

Recent years have seen rapid growth in the area of decision-support queries. Such queries typically operate over huge amounts of data (many Terabytes), performing multiple joins and complex aggregation. Furthermore, they are becoming increasingly more interactive, requiring response times in the order of seconds. Traditional optimization techniques often fail to meet these new requirements. In such cases, a solution often used in practice is to create a number of *materialized views* (MVs) that contain the pre-computed results of the common operations in a set of user queries; individual user queries can then be optimized by accessing the MVs instead of the raw data.

Permission to make digital or hard copies of part or all of this work or personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
MOD 2000, Dallas, TX USA
© ACM 2000 1-58113-218-2/00/05 . . .$5.00

In this paper, we present an algorithm that is able to take advantage of MVs by proving that the contents of an MV and a user query overlap, and compensating for the non-overlapping parts. When an overlap exists, we say that the query and the MV *match*. After discovering a match, the query can be rewritten so that it will access the MV instead of one or more of the base tables. We consider MVs that are expressed as SQL queries with aggregation. Given their transparent (automatic) use in optimization, and the fact that they summarize the raw data via aggregation, we refer to such MVs as *Automatic Summary Tables* (ASTs). Experience with the TPC-D benchmark and several customer applications has shown that ASTs can often improve the response time of decision-support queries by orders of magnitude. Such performance advantages have made ASTs indispensable in data warehousing environments. Of course, for a complete AST solution, the following related problems must also be addressed: (a) finding the best set of ASTs for each workload under space and/or update overhead constraints, (b) deciding whether an AST should actually be used in answering a query, and (c) maintaining the ASTs efficiently when the base tables are updated. Examples of existing work on these problems include [7], [2], and [10] for (a), (b), and (c), respectively.

In the remainder of this section we describe a sample DB schema, give an example of matching, and list our main contributions. In Section 2, we briefly explain the query graph model. In Section 3, we describe a general matching infrastructure, and then, in Sections 4 and 0 we present a number of specific matching "patterns". In Section 6, we give the details of how individual expressions from a query are matched with (or derived from) the expressions of an AST. In Section 7, we review related work. Finally, in Section 8 we summarize the main points of the paper.

1.1 Sample Database and Example

Figure 1 shows the sample DB schema that we have used for the examples in this paper. The schema contains a single fact table (Trans), which records credit card transactions. Each transaction corresponds to the purchase of one product, and records the product group (fpgid), the location (flid) and date of the purchase, the credit card id (faid), the

number of items purchased (qty), and the price and discount rate for the product. The product groups, locations, and credit card accounts comprise three of the schema's dimensions. The product dimension consists of a single level recorded in the PGroup table. The location dimension contains city, state, and country levels, and is represented by a single, de-normalized table (Loc). The account dimension contains two levels, represented by the Cust and Acct tables. The schema contains a Time dimension as well, which is encoded in the date field of the Trans table. The Time levels (day, month, and year) are extracted from the date field using built-in functions. The arrows in Figure 1 represent referential integrity (RI) constraints connecting the fact table to the dimensions.

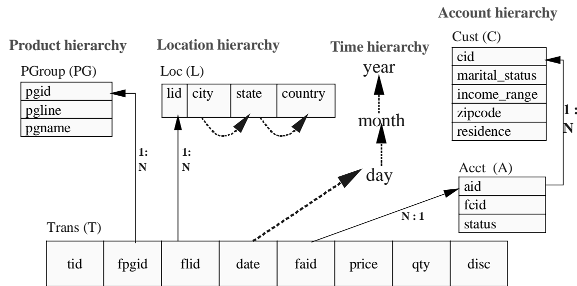


Figure 1: Simplified credit card schema

```
Q1: select faid, state, year(date) as year,
          count(*) as cnt
    from Trans, Loc
    where flid = lid and country = 'USA'
    group by faid, state, year(date)
    having count(*) > 100

AST1: select faid, flid, year(date) as year,
            count(*) as cnt
      from Trans
      group by faid, flid, year(date)

NewQ1: select faid, state, year, sum (cnt) as cnt
       from AST1, Loc
       where flid = lid and country = 'USA'
       group by faid, state, year
       having sum (cnt) > 100
```

Figure 2: Example of query rewrite

An analyst of such an application will be interested in aggregating the transaction data along different dimensions and levels. For example, query Q1 in Figure 2 counts the number of transactions performed in USA per each account, state, and year and returns the counts that are greater than 100. If AST1 is defined as in Figure 2, then Q1 can be rewritten as NewQ1, which accesses AST1 instead of the Trans table. Given that the average customer performs a few hundred transactions per year, most of them within the same city, AST1 is about hundred times smaller than Trans. Therefore, NewQ1 should perform much better than Q1.

1.2 Our Contributions

The problem of query matching has received considerable attention before. However, previous work has focused on certain simple query patterns only. In our work, we have build on such simple patterns in order to design a powerful algorithm that extends prior art in the following ways.

1. Modern SQL applications make heavy use of subqueries. Today, scalar subqueries can be used wherever a scalar is expected, and subqueries that return tables can be used wherever a table is expected. This flexibility leads to complex multi-block queries. In the best scenario, existing algorithms can rewrite only the innermost query blocks using single-block ASTs. In contrast, the algorithm described here can match multi-block queries with multi-block ASTs.
2. Even in the context of single-block queries, existing algorithms cannot handle complex expressions very well. Often, SELECT and GROUP-BY lists are restricted to base table columns only or aggregate functions of base table columns. Here, we extend the existing work to include arbitrary expressions.
3. Multidimensional aggregation (expressed via *supergroup functions* like cube, rollup, and grouping sets) is fundamental in decision-support applications. Nevertheless, it has not been considered before in the context of matching. Here, we present matching conditions for queries and ASTs that perform multidimensional aggregation.
4. Previous algorithmic descriptions have often been rather abstract. Sometimes the emphasis is on theoretical results, and at other times, important details are omitted. In contrast, here we describe in detail a practical algorithm, a substantial portion of which has been implemented inside DB2 UDB. Furthermore, the algorithm is modular and extensible; it consists of a generic matching infrastructure and a collection of matching conditions for specific query patterns.

2. BACKGROUND: THE QGM MODEL

In this section, we give a brief overview of the Query Graph Model (QGM), which serves as the basis for our matching algorithm. In QGM, a query is represented as a rooted directed acyclic graph[1] in which the leaf nodes (boxes) represent base tables, internal nodes represent table operations, and edges represent a flow of records from a child (producer) box to a parent (consumer) box. Each non-leaf box produces a relational table after performing its operation on its input, which is a set of relational tables. The root QGM box produces the final query result.

QGM boxes are labeled by the *type* of their operation. The two most common types are SELECT and GROUP-BY. SELECT boxes represent the select-project-join portions of queries; they apply the WHERE or HAVING predicates, and compute all of the scalar expressions that appear in SELECT and GROUP-BY clauses. GROUP-BY boxes perform grouping and compute the aggregate functions. For example, the QGM graph for query Q1 is shown in Figure 3. The bottom SELECT box in the figure performs a join

[1] In this paper, we will not consider correlated or recursive queries whose QGM graphs contain cycles.

between the Trans and Loc tables, as specified by the flid = lid join predicate, applies the selection predicate country = USA to the records coming from the Loc table, computes the year(date) grouping expression, and passes on the values of this expression as well the values of the faid and state columns to the parent GROUP-BY box. The GROUP-BY box groups its input records by faid, state, and year, computes the number of records per group, and passes on the grouping columns and the counts to its parent. Finally, the top SELECT box applies the HAVING predicate cnt > 10 and exports the final result. It should be emphasized that QGM represents the query semantics and not any particular execution plan. For example, the bottom SELECT box in Figure 3 does not dictate whether the join is performed before or after the selection of the USA locations.

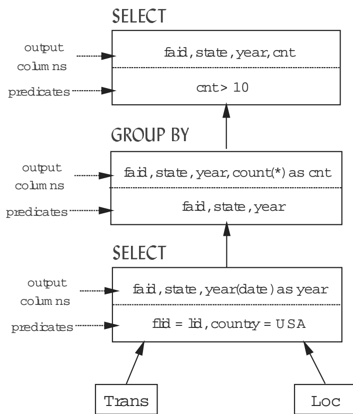


Figure 3: QGM graph for query Q1

As indicted by Figure 3 and the above discussion, a QGM box is described by its type and by its *input columns*, *output columns*, and *predicates*. The predicates and the output columns are computed by *expressions* that are built using input columns. In the remainder of this section, we define these constructs in greater detail.

The *input columns* (or *QNCs*, in QGM terminology) of a box are the columns consumed by the box; their values are produced by the children of the box and flow along the edges that connect the box to its children. QNCs are used, together with functions, operators, and constants, to build *expressions*. Expressions that consist of a single QNC or constant are considered *simple*; otherwise, they are *complex*. Expressions specify the computations for the output columns and the predicates of a box. *Predicates* are found in both SELECT and GROUP-BY boxes. SELECT predicates may be simple selection predicates, join predicates, or selection predicates with subqueries. As a result, a SELECT box may have multiple children, which are join operands or subqueries. GROUP-BY predicates describe the groups to be created. Such grouping predicates are either simple QNCs (like faid, state, and year in Figure 3) or supergroup functions over simple QNCs. GROUP-BY boxes have a single child always. The o*utput columns* (or *QCLs*, in QGM terminology) of a box are the columns produced by the box itself. For SELECT boxes, QCL expressions can be arbitrarily complex as long as they do not contain any aggregate functions. The reverse is true for GROUP-BY boxes; their QCLs include all of the grouping input columns, plus aggregate functions over simple input columns. (Given that all of the grouping QNCs are QCLs as well, we refer to such columns simply as *grouping columns*. The set of these grouping columns is the *grouping set* of the box.) It should be noted that a given QCL may be consumed by multiple parent boxes, and hence, there is a 1:N relationship between QCLs and QNCs.

3. THE MATCHING FRAMEWORK

As explained below, the matching algorithm is based on the idea of matching pairs of QGM boxes. In general, a box E *matches* with another box R, if and only if a QGM graph G(E,R) can be constructed such that G(E,R) contains the subgraph G(R) rooted at R, and G(E,R) is semantically equivalent to the subgraph G(E) rooted at E, i.e., G(E,R) and G(E) always produce the same result. If box E matches with box R, then G(E,R) – G(R) is the *compensation*, that is, the set of operations that have to be performed on the output of R in order to get the same output as E. A graphical representation of this definition is shown in Figure 4. If the compensation is empty, the match is *exact* and boxes E and R are equivalent; otherwise, box E is equivalent to the root box of the compensation. Obviously, a non-exact match relationship is asymmetric; to distinguish the different roles of the two boxes in such a relationship, we call E the *subsumee* and R the *subsumer*.

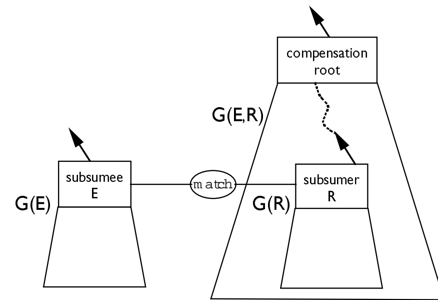


Figure 4: The matching relationship

Having defined the match relationship, we can now present the matching algorithm, starting with its two main components: the *match function* and the *navigator*. The match function takes as input two QGM boxes and determines whether they match. Ideally, the match function should implement the match relationship defined above. However, this definition is too general for practical use. In reality, the match function tries to approximate the match definition in meaningful and practical ways. It does so by considering certain simple, but general graph *patterns*,

which consist of the given subsumee and subsumer, as well as the compensation boxes for the matches between the children of the subsumee and subsumer. For each pattern, the match function tests a number of sufficient conditions to determine if a match is possible. Two such conditions that are common for every pattern are the following:

1. At least one of the subsumee's children must match with some subsumer child.
2. The subsumee and subsumer must be of the same type.

The first condition makes sure that there is some minimum overlap between the two boxes. The second condition serves as a quick test, although it is somewhat restrictive[2].

The match function is driven by the navigator. The navigator scans the query and AST graphs in a bottom-up fashion, identifying potential pairs of matching boxes (where the subsumee comes from the query graph and the subsumer comes from the AST graph), and invoking the match function, until the root AST box is matched (if possible) with one or more query boxes. To perform its task, the navigator initializes a set of candidate subsumee/ subsumer pairs by forming all of the pairs between the leaves of the graphs. During each iteration, the navigator removes a pair from this set and passes it to the match function. If a match is established, the navigator forms all of the pairs between the parents of the subsumee and the subsumer. The navigator processes its set of candidate box pairs in an order that guarantees that during each invocation of the match function, the children of the two input boxes have been matched already, i.e., the match function has been invoked on each pair-wise combination of the children. Furthermore, the match function knows the compensations for the matches between the children. As a result, it does not have to look at the whole subgraphs of its input boxes; it needs to concentrate on the subsumee, the subsumer, and the child compensation boxes only.

4. MATCH FUNCTION PATTERNS

In this section, we present a list of patterns for matching. The patterns listed here consist of SELECT and/or *simple* GROUP-BY boxes, i.e., GROUP-BYs with no supergroup functions. We start by considering patterns where all of the matches among the children of a candidate subsumee/ subsumer pair are exact. Then we present patterns where the child matches have compensation. In each case, we first state the matching conditions, then describe the compensation, and finish with an example. It should be noted that the matching conditions are sufficient only, and as a result, they are correct only when viewed together with the associated compensation. Due to space, correctness proofs are not included here; instead, some intuitive justification is given in the context of the examples.

Before we proceed, some more terminology must be established regarding the children of two SELECT boxes in a candidate match. As we will see, it is possible to have a subsumee child that does not match with any of the subsumer children; such a subsumee child is called a *rejoin* child. It is also possible to have a subsumer child with no matching subsumee child; such a subsumer child is called an *extra* child and a join between an extra child and the rest of the subsumer is called an *extra join*.

4.1 Exact Child Matches

4.1.1 SELECT boxes with one-to-one child matches

Pattern: The subsumee and subsumer are SELECT boxes and (a) each subsumee child matches with at most one subsumer child, (b) no two subsumee children match with the same subsumer child[3].

Matching Conditions: (1) Every extra join is lossless, i.e., it does not duplicate or eliminate any subsumer rows. (2) Every subsumer predicate that is not an extra join predicate is semantically equivalent (matches) with some subsumee predicate[4]. (3) Every subsumee predicate matches with a subsumer predicate or is derivable from the subsumer's QCLs and/or the QCLs of the rejoin children (if any). (4) Each subsumee QCL is derivable from the subsumer's QCLs and/or the QCLs of the rejoin children. (A subsumee expression (predicate or QCL) is derivable, if it can be written as a function of the subsumer and/or the rejoin QCLs. The details about expression equivalence and derivability are given in Section 6.)

Compensation: The compensation consists of the rejoin children (if any) and a SELECT box that (a) rejoins the subsumer with the rejoin children, (b) applies all of the subsumee's predicates that do not have matching subsumer predicates, and (c) derives all of the subsumee's QCLs from the subsumer's QCLs and/or the rejoin QCLs.

Example: Figure 5 shows a match between query Q2 and AST2. The QGM graphs for Q2 and AST2 consist of one SELECT box joining three base tables. As explained below, the two SELECT boxes satisfy all of the above conditions, and hence they match with a compensation that consists of a SELECT box (Sel-1C1) and the PGroup table. The rewritten query is NewQ2 in Figure 5. In this example,

[2] For example, a SELECT DISTINCT box may match with a GROUP-BY box, as they both eliminate duplicates. A way to match SELECT DISTINCT and GROUP-BY without violating condition 2 is presented in [13].

[3] These assumptions are not always true. (Usually, they are violated when self-joins are involved.) A method for relaxing these assumptions is described in [13].

[4] More generally, every subsumer predicate must "subsume" some subsumee predicate, where p1 subsumes p2 if every row eliminated by p1 is also eliminated by p2. For example, x > 10 subsumes x > 20.

PGroup is a rejoin child and Loc is an extra child. Condition 1 is satisfied, as the RI constraint between columns flid and lid makes the join between Trans and Loc lossless. For condition 2, the relevant subsumer predicates are faid = aid and disc > 0.1, both of which appear in the subsumee as well. As a result, the AST does not eliminate any rows that are needed by the query. For condition 3, the relevant subsumee predicates are fpgid = pgid, price > 100, and pgname = TV, all of which are derivable. As shown in Figure 5, these predicates become part of the compensation. With respect to QCL derivability (condition 4), two things are worth observing. First, the compensation derives Q2's aid column from the AST's faid column. Although aid and faid originate from different base tables, they are equivalent because of the faid = aid join predicate. Our algorithm is able to recognize such column equivalence and thus derive aid from faid. Second, the amt column can be derived from the AST using the qty, price, and disc QCLs, or the disc and value QCLs. As shown in Figure 5, when alternative derivations are possible, we choose the one that involves the minimum number of subsumer QCLs.

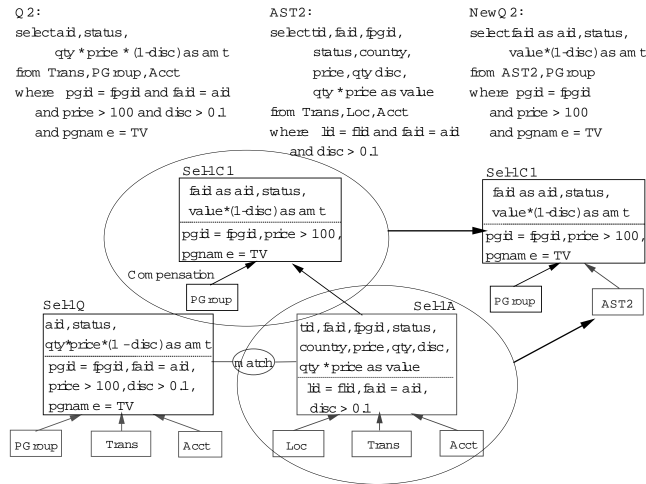


Figure 5: SELECT boxes with exact child matches.

4.1.2 GROUP-BY boxes

Pattern: The subsumee and subsumer are simple GROUP-BY boxes whose children match exactly.

Matching Conditions: (1) Every subsumee grouping column is semantically equivalent (matches) with some subsumer grouping column. (2) If the subsumee's and subsumer's grouping sets match exactly, i.e., every subsumee grouping column matches with a subsumer grouping column and vice-versa, then every aggregate subsumee QCL matches with some subsumer aggregate QCL; otherwise, every aggregate subsumee QCL is derivable from the subsumer's QCLs.

Compensation: No compensation is required if the subsumee and subsumer grouping sets match exactly. Otherwise, the compensation consists of a GROUP-BY box that re-groups by the subsumee's grouping columns, and derives the subsumee's QCLs from the subsumer's QCLs. For aggregate functions, special derivation rules must be observed; these are listed below for the most common aggregates. (The rules can be combined to derive any other aggregate that is an algebraic expression of the listed functions.) Throughout this list we assume that x is a subsumee QNC, y and z are subsumer QNCs, z is non-nullable, and x and y are semantically equivalent.

a. COUNT(*) is derived as SUM(cnt), where cnt is the COUNT(*) subsumer QCL or the COUNT(z) subsumer QCL.
b. COUNT(x) is derived as SUM(cnt), where cnt is the COUNT(y) subsumer QCL. If x is non-nullable, then cnt might also be the COUNT(z) subsumer QCL.
c. SUM(x) is derived as SUM(sm), where sm is the SUM(y) subsumer QCL. If y is a grouping column, then SUM(x) can also be derived as SUM(y*cnt), where cnt is the COUNT(*) subsumer QCL; in this case, the compensation includes a SELECT box as well to compute the y*cnt expression before regrouping.
d. MAX(x) is derived as MAX(max) or MAX(y). In the first derivation, max is the MAX(y) subsumer QCL; in the second derivation, y must be a grouping column.
e. MIN(x) is similar to MAX(x).
f. COUNT(distinct x) is derived as COUNT(y), if y is a grouping column.
g. SUM(distinct x) is derived as SUM(y), if y is a grouping column.

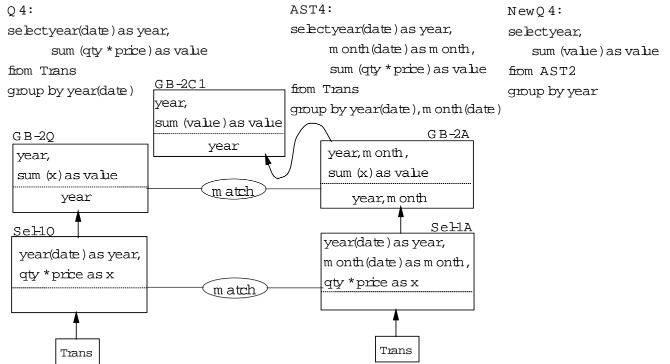


Figure 6: GROUP-BY boxes with exact child matches

Example 1: In Figure 6 the two SELECT boxes (Sel-1Q and Sel-1A) are matched first using the conditions from Section 4.1.1. Sel-1Q and Sel-1A match exactly[5]. As a result, the two GROUP-BY boxes (GB-2Q and GB-2A)

[5] Strictly speaking, the match is not exact because Sel-1A produces more columns than Sel-1Q. However, if the only difference is that the subsumer produces more columns than the subsumee, then we consider the match to be exact, unless the subsumee is the top query box in which case compensation is required to project out the extra subsumer columns.

comply with the current pattern, and are matched using the current conditions. This match requires re-grouping compensation (box GB-2C1) in order to compute the query's yearly groups from the AST's monthly groups. Additionally, the compensation derives the yearly sums by re-summing the monthly sums, using rule (c) above. This derivation is correct because the AST's monthly sums are partial sums for the query's yearly sums.

4.2 Non-Exact Child Matches

When the children of a given subsumee/subsumer pair do not match exactly, then, in addition to the subsumee and subsumer, we must also consider the boxes that comprise the compensations for the non-exact child matches. Usually, these child compensation boxes have to be included in the compensation for the parent match. This is called *pulling up* the child compensation boxes.

4.2.1 GROUP-BY boxes with SELECT-only child compensation

Pattern: The subsumee and subsumer are GROUP-BY boxes whose children match with compensation that is a single[6] SELECT box, which may perform rejoins. Furthermore, we assume here that if AGG(x) is a subsumee aggregate function, then QNC x originates from non-rejoin columns only (this assumption is relaxed in [13]).

Matching Conditions: (1) Every subsumee grouping column is derivable from the subsumer grouping columns and/or the rejoin QCLs (if any). (2) If no regrouping compensation is required, then every subsumee aggregate QCL matches with some subsumer aggregate QCL. Otherwise, every subsumee aggregate QCL is derivable from the subsumer's QCLs. (3) Pullup condition: every predicate in the child compensation is derivable from the subsumer's grouping columns and/or the rejoin QCLs.

Compensation: The compensation includes the pulled up SELECT box, potentially followed above by a GROUP-BY box. If the child compensation does not perform rejoins, then the rule for including or not the GROUP-BY box is the same as in Section 4.1.2. Otherwise, regrouping can be avoided only if the two grouping sets are the same **and** the rejoin is 1:N with the rejoin tables being the "1" side. If regrouping is required, then the aggregate functions are derived using the rules of Section 4.1.2 again.

Example1 (no rejoins): In Figure 7, the two SELECT boxes (Sel-1Q and Sel-1A) are matched first, creating the Sel-1C1 compensation box, which comprises the child compensation for the next match between the two GROUP-BY boxes. GB-2Q and GB-2A satisfy all the conditions of the current section, and as a result, they match with a compensation that consists of boxes Sel-2C1 and GB-2C2. Sel-2C1 is the pulled-up version of Sel-1C1. It is worth observing that Sel-1C1 is not pulled up "as is"; as indicated by the pullup condition, only the predicates are pulled up. In contrast, the QCLs that appear in Sel-2C1 are created there as a side effect of deriving the subsumee's expressions (see Section 6). The reasoning behind this tactic of not pulling up the QCLs can be explained in the context of the "x" QCL: x is needed in Sel-1C1 to make that box equivalent to Sel-1Q. However, x is not preserved at the output of the parent GB-2Q box; it is used there internally only, to compute the sums. Furthermore, sum(x) is derived from the AST as sum(value). As a result, what we need in Sel-2C1 is value, not x. This pullup tactic is not unique to this pattern; it is used whenever compensation is pulled up. With respect to predicate pullup, we notice that the AST rows eliminated by the month > 6 predicate in Sel-2C1 are exactly the same rows (modulo duplicates) that are eliminated by the same predicate in Sel-1Q. As a result, the predicates in Sel-2C1 and Sel-1Q have the same effect. Finally, as in Section 4.1.2, condition 1 of the current section guarantees that each subsumer group is a partial group of exactly one subsumee group. As a result, re-grouping and re-aggregating in GB-2C2 produces the correct result.

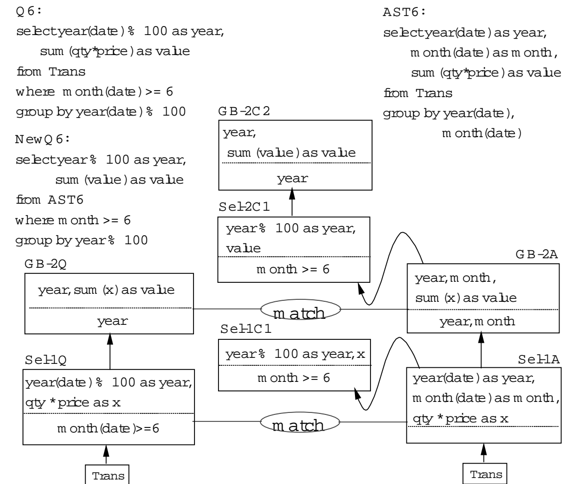
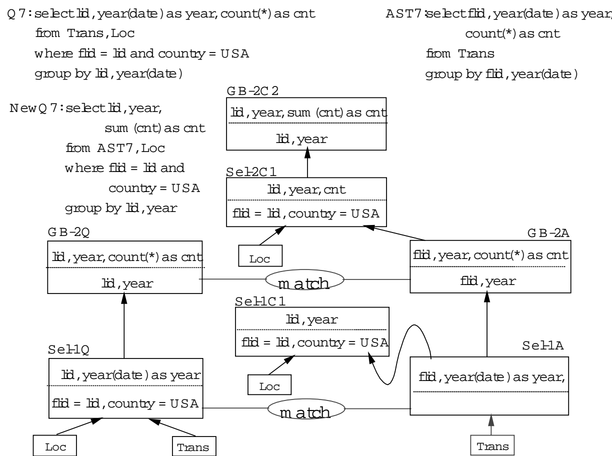


Figure 7: GROUP-BY boxes with simple SELECT child compensation

Example2 (with rejoin): Figugre 8 shows an example with rejoins. Let's assume, for the moment, that the join between Loc and Trans in Q7 is an N:M join. Then, according to the above rules, the compensation between the two GROUP-BY boxes (GB-2Q and GB-2A) must include a GROUP-BY box (GB-2C2). To see why this compensation is correct, we first observe that Q7 joins Trans with Loc, whereas NewQ7 joins AST7 with Loc. Given that AST7 is a summarization of Trans, the outputs of Sel-1Q and Sel-2C1 differ only in the multiplicity of their rows; Sel-1Q produces more duplicates than Sel-2C1. Other than

[6] The assumption that the child compensation consists of a single SELECT box is not restrictive because consecutive SELECT boxes can (almost) always be merged into a single SELECT.

duplicates, however, Sel-1Q and Sel-2C1 produce the same rows. Furthermore, Sel-2C1 “remembers” the number of the lost duplicates in its cnt QCL. As a result, Sel-2C1 does not lose any information, and the counts computed by the query can be derived in GB-2C2 by re-grouping and summing over the AST’s cnt column. If we take into account the fact that the join between Loc and Trans is 1:N, then GB-2C2 is not needed because the join does not affect the multiplicity of the Trans rows. Furthermore, the effect of the country = USA predicate is to eliminate some whole groups, but it does not affect the number of Trans rows that fall into each group. Hence, the counts produced by Q7 are the same as the counts produced by AST7.



Figugre 8: GROUP-BY boxes with rejoin child compensation

4.2.2 *GROUP-BY boxes with GROUP-BY child compensation*

Pattern: The general form for this pattern is shown in Figure 9. The subsumee and subsumer are GROUP-BY boxes (GB-Q and GB-A) and the child compensation contains at least one GROUP-BY box and a number (possibly zero) of SELECT boxes. In Figure 9, GB-cC2 is the lowest GROUP-BY box in the child compensation.

Matching Condition: To handle this pattern, the match function calls itself recursively, trying to match GB-cC2 with the subsumer (GB-A). This recursive invocation of the match function conforms to patterns 4.1.2 or 4.2.1: GB-cC2 plays the role of the subsumee, GB-A is the subsumer, and Sel-cC1, if present, is the child compensation. If this intermediate match succeeds, then the original match (between GB-Q and GB-A) succeeds as well.

Compensation: To build the compensation, we start with the intermediate compensation for the match between GB-cC2 and GB-A. Then, all the child-compensation boxes above GB-cC2 are copied above GB-pC2 in the parent compensation. For example, Box-pCN is an exact copy of Box-cCN. Finally, the original subsumee (GB-Q) is also copied at the top of the parent compensation (GB-pC(N+1)). To see why this construction is correct, we first notice that if GB-cC2 and GB-A match, then, by the match definition, GB-cC2 and GB-pC2 are equivalent. As a result, all boxes above GB-cC2 in the child compensation are equivalent to their copies in the parent compensation. In particular, Box-cCN is equivalent to Box-pCN. However, Box-cCN is also equivalent to child-Q, due to the match between child-Q and child-A. We conclude that child-Q is equivalent to Box-pCN. As a result, boxes GB-Q and GB-pC(N+1) are equivalent as well, because they are copies of each other and their children are equivalent.

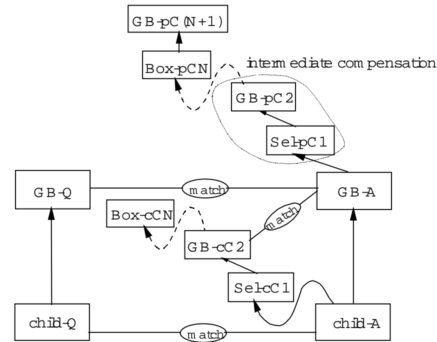


Figure 9:GROUP-BY boxes with GROUP-BY child compensation (general form)

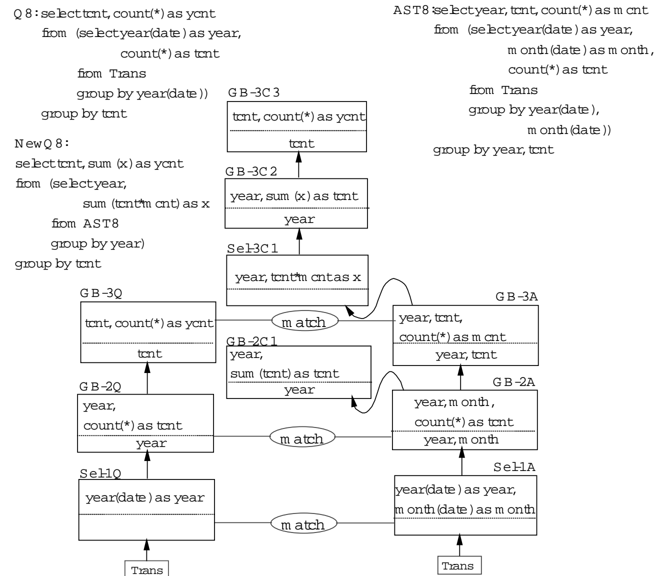


Figure 10: GROUP-BY boxes with GROUP-BY child compensation

Example: In the example of Figure 10, Q8 and AST8 are histogram queries; Q8 prints out all of the yearly transaction counts, and for each such value it gives the number of different years during which that count was achieved. AST8 performs the same computation but for monthly transaction counts. In Figure 10, box GB-2C1 is the compensation for the match between the two inner GROUP-BYs. Boxes GB-3C2 and Sel-3C1 is the compensation for the match

between GB-2C1 and GB-3A, where the conditions and rules of Section 4.1.2 were used. Finally, box GB-3C3 is a copy of GB-3Q that completes the compensation for the match between GB-3Q and GB-3A.

4.2.3 SELECT boxes with SELECT-only child compensation

Pattern: The subsumee and subsumer are SELECT boxes whose children match with compensations that do not include any grouping.

Matching Conditions: The conditions here are similar to the ones in Section 4.1.1, but adjustments have to be made to include the child compensation boxes. The revised conditions are: (1) Same as in 4.1.1 (2) Every subsumer predicate that is not an extra join predicate matches with (or subsumes) some subsumee or child compensation predicate. (3) Same as in 4.1.1. (4) Same as in 4.1.1. (5) Pullup condition: Every child compensation predicate that does not have a matching subsumer predicate is derivable from the subsumer's QCLs and/or the rejoin QCLs (if any).

Compensation: It includes the rejoin children (if any) and a single SELECT box that contains all the subsumee and/or child-compensation predicates that do not have matching subsumer predicates.

Example: An example is given in [13].

4.2.4 SELECT boxes with GROUP-BY child compensation, but no common joins

Pattern: The subsumee and subsumer are SELECT boxes with no overlapping joins and at most one child match whose compensation includes grouping.

Matching Conditions: The matching conditions are the same as in Section 4.2.3, with the addition of a pullup condition for the GROUP-BY box(es) of the grouping child compensation: every predicate (i.e., grouping column) of every child-compensation GROUP-BY box is derivable from the subsumer and/or the rejoin QCLs.

Compensation: The compensation is built in three steps. First, any non-grouping child compensations are pulled up as described in Section 4.2.3. This creates a single parent-compensation SELECT box (call it Sel-pC1). Then, the grouping child compensation is pulled up on top of Sel-pC1. Finally, another SELECT is added at the top to compensate the subsumee'spredicates and QCLs.

Example: Figure 11 shows an example for this pattern (the bottom SELECT boxes have been omitted to save space). In this example, the two top SELECT boxes (Sel-3Q and Sel-3A) are matched using the conditions for the current pattern. Compensation boxes GB-3C2 and Sel-3C1 are the pulled-up versions of GB-2C2 and Sel-2C1, respectively. Box Sel-3C3 is the additional SELECT box inserted to compensate the subsumee's cnt > 2 predicate and derive its QCLs. This example is discussed further in Section 6.

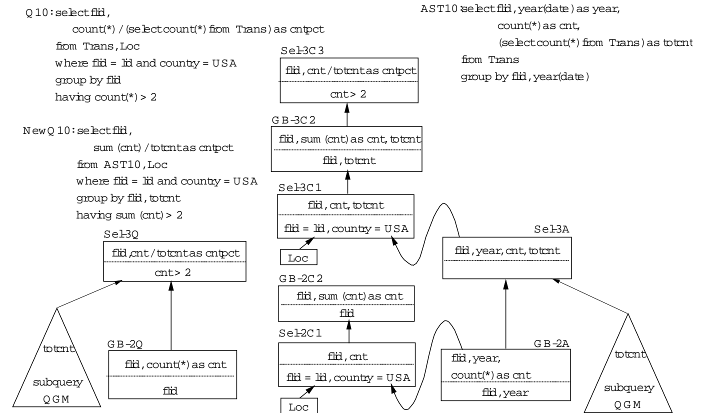


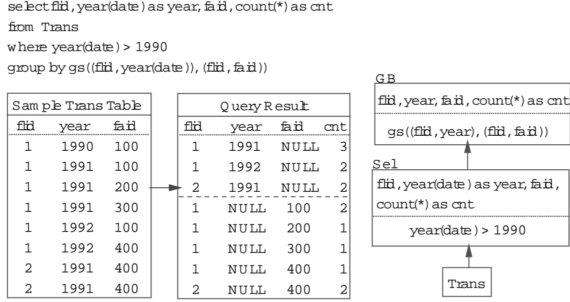
Figure 11: SELECT boxes with GROUP-BY child compensation

5. MATCHING CUBES

Recently, SQL has been extended with the introduction of three *supergroup* functions – rollup, cube, and grouping sets (or "gs" for brevity) – which allow multiple simple group-by queries to be expressed within a single SQL statement. Specifically, multidimensional grouping can be expressed by a GROUP-BY clause that contains any combination of the supergroup functions, e.g., group by rollup(a,b), gs((f,g), (f,h)). It turns out that every supergroup expression can be converted to an equivalent *canonical* expression that consists of a single gs function: gs(GS_1, GS_2, …, GS_k), where each GS_i is a simple grouping set [9]. In this section, we present matching conditions for such canonical expressions only[7]. We start by explaining the precise semantics of the gs function. Then, we present two patterns with multidimensional group-by's.

Let Q be a query block with the following GROUP-BY clause: group by gs(GS_1, GS_2, …, GS_k), where $GS_i = \{ A_1^i, A_2^i, \ldots, A_{n_i}^i \}$. Let $GS = \bigcup_1^k GS_i$, and N be the number of elements in GS. Then Q is equivalent to the union of k simple group-by query blocks, known as *cuboids*, and Q is said to be a *cube query*. Each cuboid Q_i groups by GS_i and produces N columns – one column for each grouping item in GS_i, plus N – n_i NULL-valued columns, that is, one NULL-valued column for each grouping item that belongs to GS but not to GS_i. An example of a cube query is shown in Figure 12. For simplicity, we assume that all of the base table columns are non-nullable, and as a result, the only NULL values appearing at the output are the ones added to represent the grouped-out columns of each cuboid.

[7] For efficiency, our algorithm matches cubes and rollups directly, without expanding them to grouping sets. However, the basic matching ideas are the same in every case.



Sample Trans Table

flid	year	faid
1	1990	100
1	1991	100
1	1991	200
1	1991	300
1	1992	100
1	1992	400
2	1991	400
2	1991	400

Query Result

flid	year	faid	cnt
1	1991	NULL	3
1	1992	NULL	2
2	1991	NULL	2
1	NULL	100	2
1	NULL	200	1
1	NULL	300	1
1	NULL	400	1
2	NULL	400	2

Figure 12: Cube query with sample result and QGM graph

5.1 Simple GROUP-BY query with cube AST

Pattern: The subsumee is a simple GROUP-BY box whereas the subsumer is a multidimensional GROUP-BY box. The child compensation may be empty or consist of a single SELECT box with or without rejoins. (Child compensations with GROUP-BY boxes are handled in exactly the same manner as in Section 4.2.2.) Let GS^E be the subsumee's grouping set, and GS_i^R, i = 1, 2, …, k be the subsumer's grouping sets.

Matching Conditions: The approach taken here is to match the subsumee with one of the cuboids that comprise the subsumer. Care must be taken, however, so that the NULL columns of a cuboid will not participate in the matching. Specifically, a match is possible if there is at least one subsumer grouping set GS_i^R such that the conditions and derivation rules of 4.1.2 or 4.2.1 are satisfied when restricted to the grouping columns of GS_i^R only (rather than all of the subsumer's grouping columns).

Compensation: If more than one of the subsumer's cuboids satisfy the matching conditions, then, to minimize the amount of regrouping in the compensation, the cuboid with the smallest number of grouping columns is selected. Let $GS_{\min}^R$ be the selected cuboid. The compensation consists of a SELECT box potentially followed above by a GROUP-BY box. The inclusion rule for the GROUP-BY box is the same as the rules in 4.1.2 or 4.2.1 restricted to the grouping columns of $GS_{\min}^R$. The SELECT box applies the pulled-up predicates from the child compensation (if any), as well as a *slicing predicate*, which selects the cuboid corresponding to $GS_{\min}^R$ out of the other cuboids. The slicing predicate is a conjunction of IS NULL and IS NOT NULL conditions over the subsumer's grouping columns: if a grouping column belongs to $GS_{\min}^R$, then it must not be NULL; otherwise it must be NULL.

Example: In Figure 13, the grouping set of Q11.1 matches exactly with the (flid, year) grouping set of AST11. In addition, rule (a) from 4.1.2 and the pullup condition from 4.2.1, restricted to (flid, year), are also satisfied. As a result, Q11.1 can be rewritten as NewQ11.1, which simply selects the (flid, year) cuboid without regrouping. In contrast, matching Q11.2 with AST11 requires regrouping, even though Q11.2 has the same grouping set as Q11.1. The reason is that the pullup condition is not satisfied in (flid, year), as this set does not include the month column. As a result, we have to use the (flid, year, month) cuboid from the AST and regroup. Finally, no match exists for Q11.3, because rule (f) from 4.1.2 requires that faid be a grouping column, and as a result, AST11 should have a grouping set with at least faid, flid, year, and month.

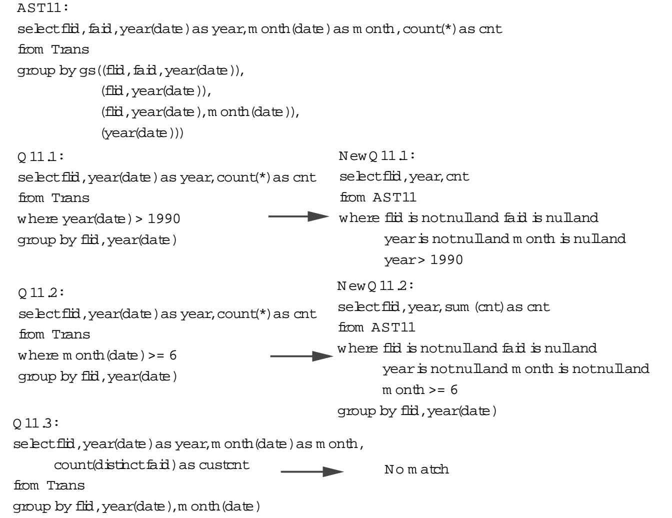


Figure 13: Simple GROUP-BY query with Cube AST

5.2 Cube Query with Cube AST.

Pattern: Both the subsumee and subsumer are multidimensional GROUP-BYs. Let GS_i^E, i = 1, 2, …, m be the subsumee's grouping sets, $GS^E = \bigcup_1^m GS_i^E$, and GS_i^R, i = 1, 2, …, k be the subsumer's grouping sets.

Matching Conditions: A match is possible if every subsumee cuboid can be independently matched with the subsumer using the conditions from Section 5.1. Otherwise, if any of these sub-matches fails, the entire match fails.

Compensation: If none of the sub-matches requires regrouping compensation, then the final compensation is a single SELECT box, which contains the pulled-up predicates from the child compensation (if any) and the slicing predicate. In this case, the slicing predicate is the disjunction of the slicing predicates for each sub-match. If, however, any of the sub-matches requires regrouping compensation, then the subsumee is treated as if it were a simple GROUP-BY, whose grouping set is GS^E. GS^E is then matched with the subsumer using the conditions from Section 5.1. In this case, the SELECT portion of the compensation is the same as in Section 5.1, i.e., it contains

the pulled-up predicates (if any) and a slicing predicate that selects the smallest subsumer cuboid that matches with GS^E. Regrouping, however, is performed not by GS^E, but by a multidimensional GROUP-BY box that has the same gs function as the subsumee.

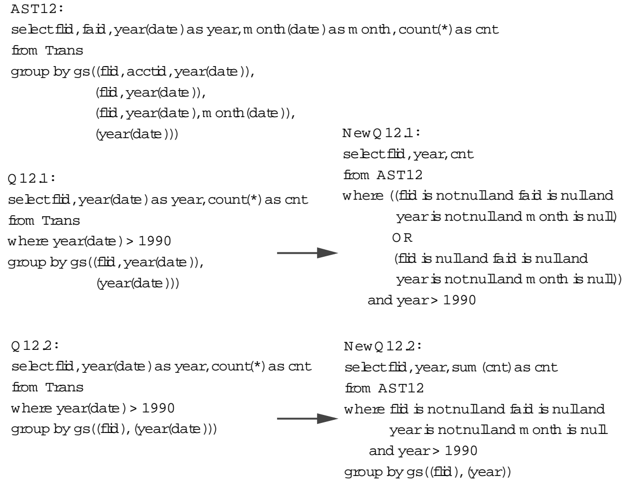


Figure 14: Matching Cubes

Example: In Figure 14, every cuboid of Q12.1 matches with AST12. None of these sub-matches requires regrouping, and hence, no regrouping is performed by NewQ12.1 either; NewQ12.1 just extracts the (locid, year) and (year) cuboids out of the AST and re-applies the year > 1990 predicate. In contrast, the match between Q12.2 and AST12 requires regrouping because the query's (locid) grouping set does not match exactly with any of the AST's grouping sets. As a result, Q12.2 is temporarily viewed as a simple group-by query, which matches with the (locid, year) AST cuboid. That cuboid is selected out of the AST in NewQ12.2, which then regroups by gs((locid), (year)).

6. EXPRESSION MATCHING AND DERIVATION

So far, we have often required that some subsumee expression E_{exp} (predicate or QCL) be semantically equivalent with some subsumer expression R_{exp}. A method is therefore required to test for expression equivalence. The first step of such a method should be to translate E_{exp} into an equivalent expression E'_{exp} that is valid within the subsumer's context, i.e., uses subsumer QNCs. This is very important because what might appear as column X inside E_{exp} may not be a direct reference to a base table column, but rather a complex sub-expression produced by a nested subquery. As a result, it is not possible to directly compare the QNCs in E_{exp} with those in R_{exp}, as they originate from different subgraphs. Once the translation is done, any expression-matching algorithm can be used to compare the parse trees of E'_{exp} and R_{exp}. However, expression matching is orthogonal to the rest of the matching algorithm, and will not be discussed further. Instead, our focus here is on the expression-translation method, which is a crucial component of our matching infrastructure.

When the subsumee and subsumer children match exactly, translation is easy. Specifically, for each QCL X produced by a non-rejoin subsumee child, there is an equivalent QCL Y produced by the subsumer's matching child. As a result, if X_E is a subsumee QNC that consumes X and appears in E_{exp}, and the subsumer consumes Y (i.e., the subsumer has a QNC Y_R), then we can replace X_E in E_{exp} with its equivalent Y_R QNC. By replacing each non-rejoin QNC in E_{exp} with its equivalent subsumer QNC (if it exists), we get E'_{exp}. Translation is more complicated when the children do not match exactly. This is best illustrated by an example. In particular, consider the example in Figure 11, but with a modified AST10 that has a HAVING predicate: count(*) > 2. Adding this predicate to AST10 makes a match between the two top SELECT boxes impossible because their predicates are not semantically equivalent, even though they are syntactically equivalent. The problem is illustrated in Table 1, which shows the query and AST results for a sample Trans table. We see that the HAVING predicate eliminates the group (1, 1991), which is necessary to produce the correct query result.

	Locid	Date	Cnt
Sample Trans Table (flid and date columns)	1	01/03/1990	
	1	02/10/1990	
	1	04/12/1990	
	1	10/20/1991	
AST Result	1	1990	3
Query Result	1		4

Table 1

Our method detects this semantic inequivalence between the two HAVING predicates by appropriately translating the query predicate. The steps taken during the translation are shown in Figure 15, where each QNC name has been annotated with the name of the box that contains the QNC. The translation begins by creating a copy of the whole expression (step 1). Then, each QNC is translated in turn. To translate a QNC, we first find the child box that produces the QNC and replace the QNC with the associated QCL expression; in our example, cnt-3Q is produced by count(*) in box GB-2Q (step 2). The next step is to replace count(*) with its equivalent QCL expression at the top of the child compensation. Thus, count(*) is translated to sum(cnt-2C2) (step 3). Then, we recursively translate each new QNC (except QNCs produced by rejoin children) until we reach the bottom of the child compensation. This way, sum(cnt-2C2) becomes sum(cnt-2C1) (step 4). Finally, we notice that cnt-2C1 and cnt-3A are equivalent, as they are both produced by the cnt QCL of the subsumer's child. As a

result, we can replace cnt-2C1 with cnt-3A (step 5). The translated predicate is sum(cnt-3A) > 2, which is obviously not the same as the subsumer's predicate cnt-3A > 2.

The translation method described above is also the first step in deriving a subsumee expression E_{exp} from the subsumer's QCLs. After translating E_{exp} to E'_{exp}, derivability can be established by making sure that the subsumer computes at its output certain necessary subexpressions of E'_{exp} (or even the entire E'_{exp}). The problem that arises, however, is to determine the parts of E'_{exp} that can/should be computed by the subsumer. In general, translation causes an expression to expand by replacing individual QNCs with equivalent subexpressions. For example, cnt-3Q > 2 is translated to sum(cnt-3A) > 2. Derivation is the reverse operation, where pieces of the translated expression are collapsed as they are computed along the derivation path. For example, sum(cnt-3A) > 2 is derived as cnt-3C3 > 2 at the top of the compensation. The next paragraph explains the derivation method in more detail in the context of the cnt / totcnt expression that computes the cntpct QCL of query Q10.

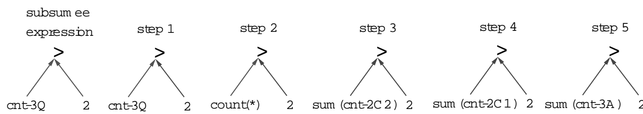


Figure 15: Expression translation

The expression is first translated as sum(cnt-3A) / totcnt-3A. During this translation, our method determines that the division operator and the sum function should be computed by the Sel-3C3 and GB-3C2 boxes, respectively. Given that those are they only internal nodes of the expression tree, and that neither of them should be computed by the subsumer, we conclude that for the expression to be derivable, the subsumer must preserve the cnt-3A and totcnt-3A QNCs at its output, which it does. As a result, the expression can be pulled up inside Sel-3C1 as sum(cnt-3C1) / totcnt-3C1. Next, the expression is pulled up one box further and becomes sum(cnt-3C2) / totcnt-3C2 within the context of GB-3C2. During this pullup, the totcnt QCL is created inside Sel-3C1 and consumed by the totcnt-3C2 QNC. The last step is to pull the expression from GB-3C2 up to Sel-3C3. To do so, the totcnt and cnt QCLs, as well as the totcnt-3C2 predicate, are created inside GB-3C2. (In fact, cnt should already be there, because it was created earlier during the derivation of the query's HAVING predicate). Notice that the cnt QCL in GB-3C2 actually computes the sum function. As a result, the expression is pulled up as cnt-3C3 / totcnt-3C3, which becomes the expression that computes the original cntpct QCL.

7. RELATED WORK

In this paper, we have presented a practical algorithm for real SQL queries with aggregation. Previous work most closely related to ours includes [6], [12], and [1]. In all three cases, the domain of the algorithms presented consists of simple, single-block select-where-groupby-having SQL statements. In [6], matching is performed by applying a set of rewrite rules to the query until a portion of it is syntactically identical to the AST. To handle multi-block queries, [6] proposed the use of similar rewrite rules for transforming multi-block queries to single-block ones. However, such a transformation is often impossible.

Another limitation of [6] is its reliance on syntactic matching, which cannot, for example, derive a query expression from the AST expressions. This limitation is also recognized in [12], where semantic matching conditions are proposed. For example, if P_Q and P_A are the sets of WHERE predicates in the query and the AST respectively, [12] requires that P_Q is equivalent to P_A & P_C, where P_C is a set of predicates that involve only AST and rejoin columns, i.e., P_C are the compensation predicates. Although this condition is more general than our conditions 2 and 3 in Section 4.1.1, no algorithm is presented in [12] for finding the P_C predicates. As we argued in Section 6, such an algorithm is orthogonal to a general matching infrastructure and should rely on first translating the query predicates into the context of the AST.

Neither [6] nor [12] make use of database semantics that are specified via constraints. Such semantics are exploited [1]. In addition to RI constraints, which have helped us handle extra AST tables, [1] considers other functional dependencies as well in order to derive query columns that are not present at the AST. Furthermore, [1] presents matching conditions for outer-join operations. In general, however, the algorithm description in [1] is rather sketchy.

Using our algorithm (or any of [1], [6], [12]), a query may be rerouted towards multiple ASTs by an iterative process where, at each iteration, the result of the previous rewrite is matched with the next available AST. A different approach is taken in [2], where, for each aggregate function, the general format of the rewritten query is determined in advance. Many candidate rewritings can be derived from the general format by simultaneously combining the aggregation results of different ASTs (e.g. sum(x) might be derived as $sum(sum_x_i*cnt_j*cnt_k)$, where sum_x_i, cnt_j, and cnt_k are computed by the ith, j^{th}, and k^{th} ASTs respectively). The candidate rewritings are then tested for equivalence with the original user query using the equivalence theory developed in [2] and [11]. Although such a global approach should, in general, be more powerful (but also less efficient) that an iterative approach, it is not clear that this is indeed the case with [2]. In particular, all of the examples

presented there can be handled by our algorithm as well. Furthermore, HAVING clauses were not allowed in [2].

Another interesting problem, from a theoretical perspective, is the discovery of *complete* rewriting algorithms, which, given a query and a set of MVs, guarantee that a rewriting will be found, if one exists. The existence and complexity of complete rewriting algorithms depends on the complexity of the queries and MVs considered and the language used to construct the rewritings. In general, finding a complete algorithm is a hard problem, and existing ones are limited to restricted classes of queries. [7] presents a complete algorithm for simple conjuctive queries without constants, comparisons, or aggregation, using set semantics. This algorithm is adapted for bag semantics in [2]. Finally, a more powerful algorithm for simple conjuctive queries and MVs that may also contain counts is developed in [4] using bag-set semantics.

8. SUMMARY

All of the related work reviewed in Section 7 applies to user and AST queries that are single block and do not contain complex expressions. In this context, queries can be described as sets of base table columns (e.g., predicate, grouping, select-list, and aggregation columns). Previous matching algorithms typically operate by comparing such sets of base table columns from the query and the AST. This approach, however, cannot handle multi-block queries and/or complex expressions, for two reasons. First, as their semantics become more complicated, queries cannot be described as single units anymore; instead they must be broken into smaller pieces, and matching should be done in a piece-by-piece fashion. Second, columns that appear in various parts of a query are not, in general, base table columns anymore; instead they are computed as complex expressions over other columns, potentially produced by nested query blocks. As a result, a translation mechanism is required before query and AST columns can be compared.

In this paper, we have presented a matching algorithm that addresses the above issues by relying on a general matching infrastructure, consisting of the QGM model, the navigator, the match function, and the translation mechanism. This infrastructure offers great modularity and extensibility by breaking the matching task into many smaller sub-matches, each involving only a small subset of QGM boxes in isolation (i.e., a subsumee, a subsumer, and their child-compensation boxes). In addition to the generic matching infrastructure, we have also presented matching conditions and compensation rules for several specific query patterns. Overall, our experience of implementing the matching algorithm inside IBM's DB2 UDB DBMS and testing its performance benefits has been very positive. Using a small number of ASTs in each case, we have seen dramatic improvements in query response times both with TPC-D queries and with a number of customer applications.

9. REFERENCES

[1] R.G. Bello, K. Dias, A. Downing, J. Feenan, J. Finnerty, W.D. Norcott, H. Sun, A. Witkowski, M. Ziauddin, "Materialized Views In Oracle", Proc. of the 24th VLDB Conf., New York, NY, 1998.

[2] S. Chaudhuri, S. Krishnamurthy, S. Potamianos, K. Shim, "Optimizing queries with materialized views", Proc. of the 11th Data Engineering Conf., Taipei, 1995.

[3] S. Cohen, W. Nutt, A. Serebrenik, "Rewriting Aggregate Queries Using Views", Proc. of the ACM-PODS Conf., Philadelphia, PA, 1999.

[4] J. Gray, A. Bosworth, A. Layman, H. Pirahesh, "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals", Proc. of the 12th Data Engineering Conf., 1996.

[5] S. Grumbach, M. Rafanelli, L. Tininini, "Querying Aggregate Data", Proc. of the ACM-PODS Conf., Philadelphia, PA, 1999.

[6] A. Gupta, V. Harinarayan, D. Quass, "Aggregate Query Processing in Data Warehousing Environments", Proc. of the 21th VLDB Conf., Zurich, Switzerland, 1995.

[7] V. Harinarayan, A. Rajaraman, J. D. Ullman, "Implementing Data Cubes Efficiently", Proc. of the ACM-SIGMOD Conf., Montreal, Canada, 1996.

[8] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, D. Srivastava, "Answering queries using views", Proc. of the ACM-PODS Conf., San Jose, CA, 1995.

[9] J. Melton (ed.), "Final Committee Draft – Database Language SQL – Part 2: Foundation (SQL/Foundation)", H2-98-519/DBL FRA-017, 1998.

[10] I. S. Mumick, D. Quass, B. S. Mumick, " Maintenance of Data Cubes and Summary Tables in a Warehouse", Proc. Of the ACM-SIGMOD Conf., Tuscon, AZ, 1997

[11] W. Nutt, Y. Sagiv, S. Shurin, "Deciding equivalence among aggregate queries", Proc. of the ACM-PODS Conf., Seattle, WA, 1998.

[12] D. Srivastava, S. Dar, H.V. Jagadish, A.Y. Levi "Answering Queries with Aggregation Using Views", Proc. of the 22nd VLDB Conf., Mumbai, India, 1996.

[13] Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, M. Urata, "Answering Complex SQL Queries Using Automated Summary Tables", available upon request from the authors.