RJ 10170 (95044) November 16, 2000
Computer Science

Research Report

APPLYING MASS QUERY OPTIMIZATION TO SPEED UP AUTOMATIC SUMMARY TABLE REFRESH

Wolfgang Lehner

University of Erlangen-Nuremberg
Martensstr. 3, Erlangen, 91058, Germany

Bobbie Cochrane
Hamid Pirahesh
Markos Zaharioudakis

IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library.CyberDig.nsf/home.

Research Division
Almaden • Austin • Beijing • Haifa • T. J. Watson • Tokyo • Zurich

APPLYING MASS QUERY OPTIMIZATION TO SPEED UP AUTOMATIC SUMMARY TABLE REFRESH

Wolfgang Lehner*

University of Erlangen-Nuremberg
Martensstr. 3, Erlangen, 91058, Germany
wolfgang@lehner.net

Bobbie Cochrane, Hamid Pirahesh, Markos Zaharioudakis

IBM Almaden Research Center
650 Harry Road, San Jose CA, 95120, U.S.A.
{bobbiec, pirahesh, markos}@almaden.ibm.com

Abstract

The real challenge in data warehousing is the optimization of aggregation queries that access a huge number of rows to retrieve aggregated summary data while performing multiple joins in the context of a typical star schema. The technique of materialized views (or 'Automatic Summary Tables', ASTs in DB2) is widely accepted as a tool to speed up these types of queries. To keep materialized views consistent with the base data, ASTs are either immediately synchronized in an incremental way or fully recomputed during a nightly refresh window. This paper addresses on the second alternative, i.e. applying mass query optimization techniques to support the refresh of multiple summary tables by extending DB2 query-matching techniques.

More specifically, the paper proposes two optimization strategies 'Query Stacking' and 'Query Sharing' for multiple summary table refresh. Stacking of query graphs may be seen as the result of common sub-expressions detection using the available query matching technology of DB2. Since exact common sub-expressions are rare, the novel optimization approach of 'Query Sharing' tries to systematically generate common sub-expressions for a given set of „related" queries, considering different predicates, grouping expressions, and sets of base tables. The theoretical framework, a prototype implementation of both strategies in the IBM DB2/UDB database system, and performance evaluations based on the TPC/R data schema are presented in this paper.

1 Introduction

Data Warehousing has inspired the database community tremendously. Many new ideas, e.g. the data cube ([GBLP96], [HaRU96]), were developed in this context over the last few years. At the same time, many old ideas (snapshots: [AdLi80], summary tables: [ChMc89], bitwise index: [ONei87], join indices: [Vald87]) have come to rejuvenation and were adapted to the warehouse application scenario (e.g. [GuMu99], [ChIo98]).

We follow the same approach and pick up the idea of Mass Query Optimization (MQO: [Jark84], [Sell88]) in the context of simultaneously re-computing (refreshing) multiple materialized views, or "Automatic Summary Tables" (ASTs) in IBM DB2/UDB terminology. Since in the traditional way of query processing, queries are in general considered isolated to each other, Mass Query Optimization was mainly applied in deductive database systems where a single change of data implies the evaluation of multiple dependent rules. Due to this restricted application area the technique never went into full blossom. Fortunately, the area of data warehousing is now providing an application context for such kind of optimization technique, for example simultaneously executing queries to re-compute materialized views.

In this paper we outline the current MQO framework of the IBM DB2/UDB database system and demonstrate the benefit of MQO by optimizing the refresh, i.e. the initial population or full recomputation, of Automatic Summary Tables. Since the refresh of a single AST may be seen as executing the query, which specifies the summary table, we have a perfect application scenario to apply MQO techniques, when issuing a single refresh command for multiple ASTs.*

1.1 Motivation of Multiple Summary Table Refresh

Since materialized views or ASTs are commonly considered a powerful tool to speed up the execution of aggregation queries, many proposals have been made to provide an automatic synchronization of the summary data in the case of updates of the underlying base data ([GuMu99]). However, such an incremental maintenance strategy might become quite expensive in the presence of many ASTs, because a single update operation on a base table implies maintenance operations on each AST ([LSPC00]). Moreover updating an AST implies keeping locks on that AST and therefore reducing the concurrency of the system. If the application, e.g. a decision support system based on a relational data warehouse, does not need fully up-to-date information, deferred refresh of ASTs is a reasonable alternative. During a nightly refresh window of a data warehouse environment, ASTs are completely recomputed, i.e. all rows of an AST are deleted, the associated database query is executed, and the result set is inserted into the AST. To beat the nightly refresh window, optimizing a set of such refresh queries and executing them simultaneously sounds attractive. Optimization is performed using our proposed query matching technology, which systematically generates and injects common sub-expressions during the rewrite phase of a refresh statement over multiple ASTs. The idea is sharing common join and group-by operations. The proposed strategy, however, is not only restricted to speed up the refresh of multiple ASTs, but can be exploited beneficially, e.g., in the following situations:

- *AST advisor:*
  The proposed strategy applied to a set of single queries reflecting a typical workload can be used to generate a set of generic candidate ASTs. Some of those candidate ASTs might be selected by the database system or the DBA, populated, and transparently used by the system to answer user queries. The approach of systematically constructing an AST has the benefit of guaranteeing that it exposes a form which is usable as a target during query rerouting.
- *Evaluation of complex grouping expressions:*
  The strategy might also be used to optimize the evaluation of queries with complex grouping expressions, especially grouping sets. Common combinations can be detected or systematically constructed.
- *Evaluation of UNION queries:*
  Many applications are still based on SQL queries using UNION to simulate CASE-expressions. Our technique can be used to detect and 'unify' such kind of 'similar' subqueries.

1.2 Contribution of the Paper

Although not restricted to this area, we demonstrate the application of two MQO strategies in the context of refreshing multiple ASTs simultaneously. In detail, the paper provides contribution in the following areas:

- The Query Graph Model of DB2 and the existing generic MQO framework used for query matching within DB2 are introduced and explained.
- The proposed optimization technique does not only recognize and utilize common sub-expressions but artificially 'adds' missing parts, thus constructively generates generic common sub-expressions. Compared to prior work, this procedure allows a much wider class of queries for simultaneous optimization and execution.
- The prototype implementation within DB2 is based on the existing query matching mechanism and extends the classical concept of (passive) query matching. Our strategy massages a given query graph, so that originally not matchable sub-queries are able to share a common sub-expression.
- The proposed techniques are independent of specific query patterns, implying that any queries (nested, correlated, ...) can be subject of optimization.

---

* The work was performed while author was visiting scientist at the IBM Almaden Research Center.

- All prior work in this context is performed in the context of the query optimizer considering physical properties, like join orders, existence of sort orders or index structures. Our approach is based on rewriting rules using higher level constructs within the Query Graph Model. Costing single alternative plans as a result of the rewrite phase by the optimizer is much cheaper than considering different alternatives within the optimizer. Moreover no approach is known exploiting information of referential integrity (RI) constraints in this context. Beyond this, it is hard for an optimizer to come up with complex compensations.
- The presentation of our approach and all performance evaluations are based on the TPC-H/R database schema ([TPC99]).
- While existing work on Mass Query Optimization focuses on the theoretical perspective, we give a detailed description on how to implement MQO techniques. We are not aware of any other work addressing this issue.

1.3 Structure of the Paper

The following section introduces the query matching framework of the IBM DB2/UDB database system. Within this section, the DB2 Query Graph Model (QGM) and necessary conditions for establishing a match between single query graphs or sub-graphs within a global query graph are outlined. Section 3 sketches the application of these matching techniques and introduces the 'Query Stacking' optimization approach. Section 4 focuses on the novel proactive query matching strategy 'Query Sharing' extending the domain of queries for simultaneous optimization. The proposed techniques of 'Query Stacking' and 'Query Sharing' are evaluated by means of experiments in section 5. Section 6 is a description and discussion of related work in the area of simultaneously optimizing multiple queries. The paper closes with a summary and a conclusion in section 7.

2 The DB2 Query Matching Framework

The DB2 query matching framework provides a sound basis for identifying common sub-expressions in query graphs. This section outlines the basic concepts of the DB2 Query Graph Model (QGM) and summarizes necessary conditions to successfully establish a match between two queries. The description is restricted to the basic entities and procedures that are necessary to understand the transformations in the 'Query Stacking' and 'Query Sharing' process. Detailed information about QGM can be found in [PiLH97]. For a detailed description of the regular matching technique in the context of transparently routing user queries to existing summary tables, we refer to [ZCP+00].

2.1 Example

To illustrate the proposed techniques, consider the following query which computes the overall quantity (sum_qty), the average discount (avg_disc), the sum of base prices (sum_base_price), and the sum of prices after discount (sum_disc_price) per order status with a quantity greater than 5 restricted to orders with a medium priority:

```
SELECT o_orderstatus
       SUM(l_quantity) AS sum_qty,
       AVG(l_discount) AS avg_disc,
       SUM(l_extendedprice) AS sum_base_price,
       SUM(l_extendedprice * (1-l_discount)) as sum_disc_price
  FROM lineitem, orders
 WHERE l_orderkey = o_orderkey
   AND o_orderpriority = '3-MEDIUM'
GROUP BY   o_orderstatus
HAVING MIN(l_quantity) > 5
```

This query (and all succeeding examples) are based on the TPC-H/R database schema ([TPC99]), where the table lineitem holds transactional data, which are evaluated according to the three 'dimensions' of orders, parts, and suppliers (figure 1).

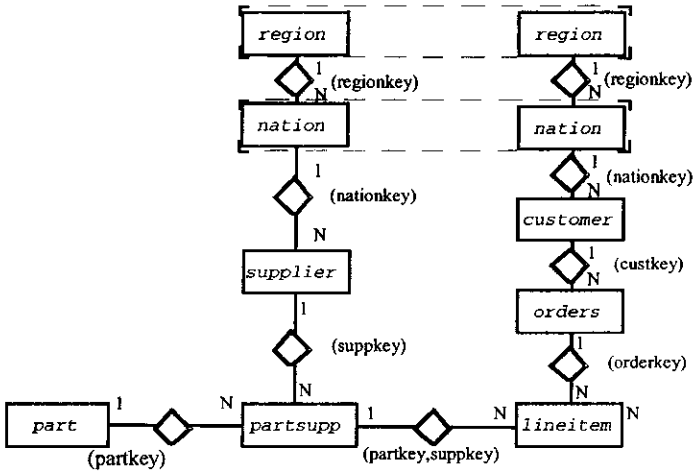


Fig. 1: ER-Diagram of TPCD-Database Schema

The order and supplier dimensions exhibit a hierarchical structure according to the location of the customer placing an order and according to the location of the supplier. It is important to mention that it is advisable to explicitly specify these 1-to-N relationships in terms of referential integrity constraints during DDL time. For example, the relationship between the lineitem and the order table is specified with a foreign key on the lineitem table as follows:

```
ALTER TABLE lineitem ADD CONSTRAINT FOREIGN KEY l_orderkey REFERENCES order(o_orderkey);
```

2.2 Overview of the DB2 Query Graph Model

In DB2, queries are parsed and internally represented by a single QGM graph. The corresponding graph for the above sample query, for example, is depicted in figure 2. In general, the QGM data structures consist of boxes which are connected by directed arcs, called quantifiers in QGM terminology. While boxes represent certain actions or classes of operations, quantifiers denote the data streams flowing from one box to another one. Within the context of this paper, we have to distinguish between three different types of boxes:

- *base table boxes:*
  Obviously base table boxes represent data sources in general and thus never exhibit incoming quantifiers. A base table box may represent a regular table, a table function, etc.
- *select boxes:*
  A select box may have multiple incoming quantifiers and may hold predicates in conjunctive normal form, which have to be applied to the data stream. Local predicates are applied to the data flow of the corresponding single quantifier, whereas non-local predicates, i.e. join predicates are applied to the set of quantifiers.
  For example, the lower select box of figure 2 contains two predicates: PRD1 for the join of the lineitem and the order table and PRD2 for the restriction on the order priority column. The select box after the group-by box computes the average aggregation function, which is not directly implemented but replaced by a division of a SUM() and COUNT() column. The uppermost select box is used to implement the having-clause of the statement. Thus it holds the predicate restricting the resulting groups to a quantity of at least 5.
- *group-by boxes:*
  As opposed to select boxes, group-by boxes do not have predicates and have exactly one incoming quantifier. However, they may either maintain a list of grouping columns, a complex grouping expression using CUBE(), ROLLUP(), or GROUPING SETS() expressions, or a combination of both ([SQL99]).
  The group-by box of the sample QGM graph of figure 2 holds the single grouping attribute O_ORDERSTATUS.

Each box may have multiple outgoing quantifiers (consumers), i.e. a specific box may be a child for more than one parent box. A box with multiple outgoing quantifiers usually represents the root of a QGM sub-graph which is considered a *common sub-expression.*

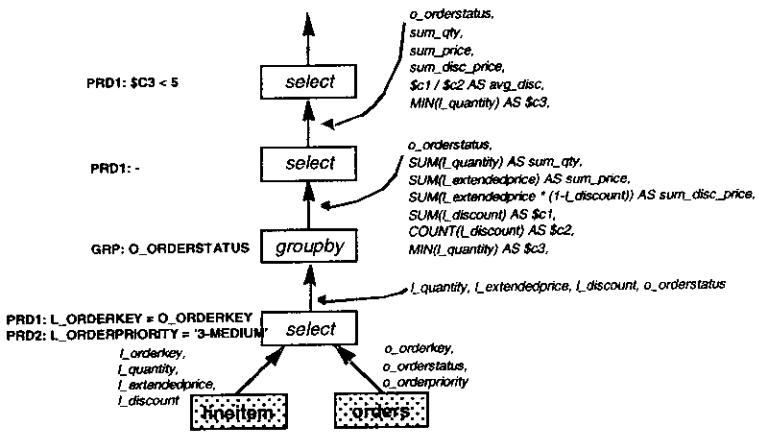


Fig. 2: Internal QGM Graph for the Sample Query

The output of a box corresponds to the set of columns which are produced by this box. Such columns may either be single columns or complex expressions which are potentially derived from multiple incoming columns, e.g. SALES*PRICE AS REVENUE. Thus each box realizes some kind of projection operator, providing only those columns that are consumed by at least one parent box. Moreover it is obvious that multiple incoming columns can contribute to a single output column and vice versa a single incoming column can appear in multiple output columns, e.g. PRICE*TAX AS X and PRICE*DISCOUNT AS Y. Additionally, a box can absorb incoming columns (for example if that column is only needed for a local predicate but not part of the output column list) or arbitrarily produce outgoing columns (for example when a single column is selected twice).

If a box has more than one outgoing quantifier then any subset of the output columns of that box can flow along any outgoing quantifier, thus splitting the producing data stream arbitrarily. For example the columns PRICE*TAX and PRICE*DISCOUNT may be used by different parent boxes.

2.3 Overview of the DB2 Query Matching Technique

To model common parts (i.e. sub-graphs) of two queries, relationships between those queries are recorded on a box-by-box level using a special QGM 'match' entity. This section summarizes the necessary conditions for recording a match (for details see [ZCP+00]).

Roles of Subsumee, Subsumer and Compensation

Typically, boxes connected by a match take on specific roles. A box is called a *subsumer*, if it provides at least all the data, which are necessary to compute the output of the corresponding *subsumee* box (figure 3). If the adjustment of the data requires some kind of additional work, then a *compensation* box is created during match recording. These compensation boxes provide the missing part successfully substituting a subsumee

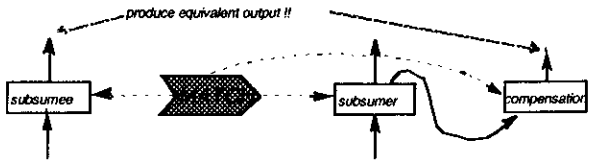


Fig. 3: General Concept of Matching with Compensation

by the corresponding subsumer. If both boxes matches exactly, matches with an empty compensation are recorded in both directions. To summarize, a match between a subsumee and a subsumer box denotes that the subsumee can be completely replaced by the compensation box based on the subsumer.

2.3.1 Matching Conditions for Box Type and Box Output

To record a match between two QGM boxes, certain conditions depending on the type of the box have to apply. Obviously it is necessary that both candidate boxes exhibit the same box type† and that base table boxes refer to the same data source.

For boxes other than base tables, a general precondition is that the expressions comprising the result of the subsumee box must be derivable from the subsumer box. Simple columns must be generated by the subsumer. Columns consisting of complex expressions must be computable from an expression of the subsumer. For example, if the subsumee has a COUNT(X) column then the subsumer also has to produce a COUNT(X) column. Note that the corresponding compensation box has to sum up the COUNT()-figures of the subsumer, in order to correctly retrieve COUNT()-figures of the subsumee.

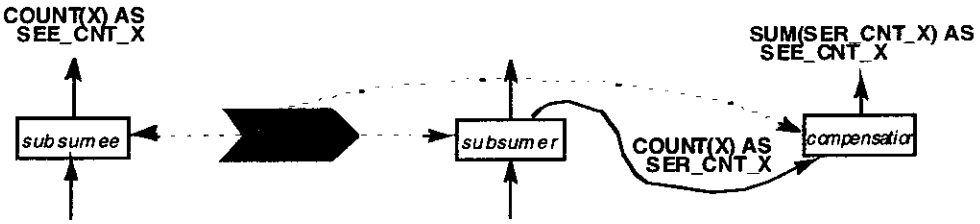


Fig. 4: Example for a Compensation of Box Output

2.3.2 Matching Conditions for Local Predicates

To record matches between two select boxes, the local predicates of the subsumer must be equal or weaker than the predicates of the subsumee. In general, the compensation box inherits all local predicates of the subsumee box that are not present in the subsumer.

The sample configurations below show three different situations where the match conditions for local predicates are fulfilled. In the first scenario, the constant of the restriction predicate is part of the IN()-list to which the subsumer is restricted. In the second scenario, the subsumer does not show a local restriction at all and in the third scenario the subsumer predicate is a disjunction where predicate subsumption is true for one alternative. If the local predicates of a subsumer and subsumee are identical, then no compensation predicate is necessary.

```
subsumee:   ... WHERE l_shipmode = 'TRUCK'
subsumer:   ... WHERE l_shipmode IN ('AIR', 'TRUCK', 'SHIP')
```

† We intentionally omit the case of establishing a match between a select box with distinct columns and a group by box.

```
subsumee:   ... WHERE l_shipmode IN ('MAIL', 'AIR')
subsumer:   ... <no predicate>

subsumee:   ... WHERE l_shipinstruct = 'DELIVER IN PERSON'
subsumer:   ... WHERE l_shipmode ='TRUCK'
                    OR l_shipinstruct IN ('DELIVER IN PERSON', 'TAKE BACK RETURN')
```

2.3.3 Matching Conditions for Joins

Join predicates of two boxes must in general match exactly and the two matching boxes must reference the same set of children. However, the existence of a loss-less join (implemented via RI constraints) allows two exceptions to this general rule.

Extra Child Compensation

The matching conditions for joins are also fulfilled, if each child table of the subsumer, which is not a child table of the subsumee is connected to the subsumer through an RI join, i.e. a lossless join. Consider the situation of figure 5, where the subsumer exhibits a join between the lineitem and orders table following the predefined primary/foreign key relationship. Since the local predicates are matching exactly, the compensation box does not even need a predicate at all.

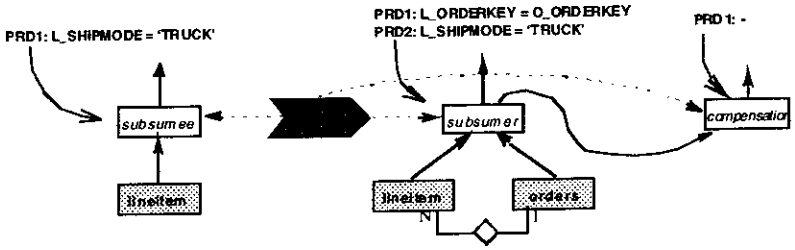


Fig. 5: Example for an Extra Child Compensation

Rejoin of Additional Children

In the second situation, each child table of the subsumee, which is not a child of the subsumer is re-joined in the compensation box, independent of the type of the join. For example, if the subsumee box shows a join between the lineitem and the orders table to query items, which were shipped on the same day on which they were ordered, the join is 'delayed' into the compensation box (figure 6). This of course requires that the necessary join column survives the subsumer.

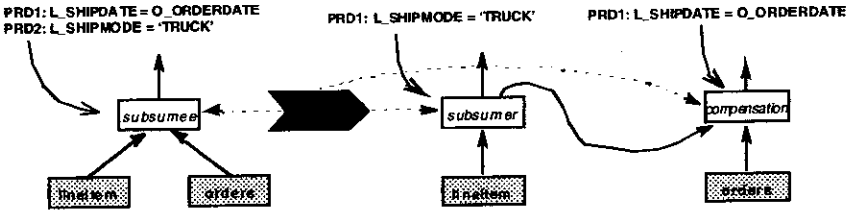


Fig. 6: Example for a Re-Join Compensation

2.3.4 Matching Conditions for Grouping Columns

To successfully record a match between two group-by boxes, the output expressions must be 'compatible' (as mentioned above) and the grouping expression of the subsumee must be derivable from the grouping expres-

sion of the subsumer. For example, since the CUBE() expression produces all possible grouping combinations ([GBLP96]), the following configuration would result in a successful match:

subsumee: ... GROUP BY l_shipmode
subsumer: ... GROUP BY cube(l_shipdate, l_shipmode)

For the remainder of the paper we restrict the presentation to simple grouping columns. The mechanism for the general case of matching any complex grouping expression is detailed in [ZCP+00]. Thus the general rule for group-by boxes in the context of this paper is that the subsumee grouping columns must be a subset of the subsumer grouping columns[‡], e.g.

subsumee: ... GROUP BY l_shipmode
subsumer: ... GROUP BY l_shipdate, l_shipmode

2.3.5 Summary of Matching Conditions

To summarize the conditions for recording box-level matches between two queries, let us refer to the refresh of the two 'Automatic Summary Tables' AST1 and AST2 of figure 7. Each AST is defined by the SQL query given in the AST definition above the corresponding QGM graph. If we consider AST1 in the role of a subsumee and AST2 in the role of a subsumer, then the lower select boxes match because both boxes have the same join predicate with regard to the set of child tables and only AST1 has a local predicate. Moreover, the extra child table of AST1, region, is joined using the primary/foreign key relationship of nation and region table. Additionally, AST2 produces a superset of columns with regard to AST1. The group-by boxes also match, because the grouping columns of AST1 are a subset of the grouping columns of AST2 and AST2 produces all aggregate function columns, which are needed for the computation of AST1.

```
CREATE SUMMARY TABLES ast1 AS (
SELECT    l_shipmode,
          n_name,
          SUM(l_quantity) AS sum_qty,
          SUM(l_extendedprice) as sum_price,
          COUNT(*) as count_order
FROM      lineitem, orders, customer, nation, region
WHERE     l_orderkey = o_orderkey
    AND   o_custkey = c_custkey
    AND   c_nationkey = n_nationkey
    AND   n_regionkey = r_regionkey
    AND   r_name = 'EUROPE'
GROUP BY l_shipmode, n_name ) ...
```

```
CREATE SUMMARY TABLES ast2 AS (
SELECT    l_shipmode, l_shipinstruct,
          n_name, n_regionkey,
          SUM(l_quantity) AS sum_qty,
          SUM(l_extendedprice) as sum_price,
          COUNT(*) as count_order
FROM      lineitem, orders, customer, nation
WHERE     l_orderkey = o_orderkey
    AND   o_custkey = c_custkey
    AND   c_nationkey = n_nationkey
GROUP BY l_shipmode, l_shipinstruct,
          n_name, n_regionkey ) ...
```

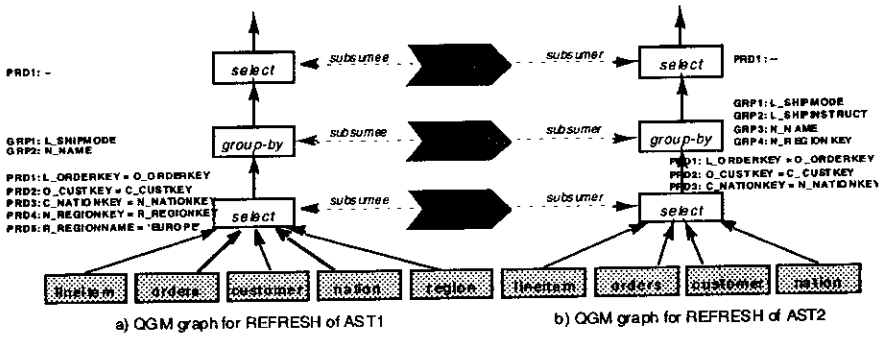


Fig. 7: Query Graphs for Refreshing two ASTs independently

‡ Actually, the set of group-by columns of the subsumer has to encompass all columns which appear as local predicates in the subsumee, too (section 4.5).

3 Query Stacking: Using Common Sub-Expressions

The first strategy, which is pursued in a first place in optimizing the execution of a multiple AST refresh is called 'Query Stacking'. The system tries to 'order' the execution of the queries and use the result of a previous query for the computation of the current and succeeding queries. This strategy may be seen as an application of the existing query matching framework of DB2. The achieved detection and utilization of common sub-expressions proves extremely powerful in the presence of aggregation operations, where aggregation dependencies can be exploited to perform 'Query Stacking'.

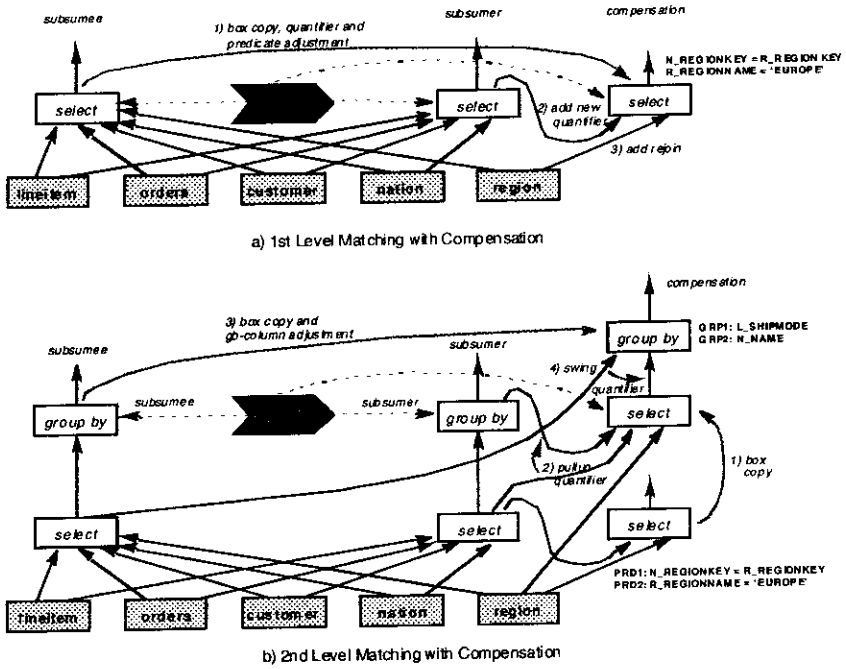


Fig. 8: Matching with Compensation

To record matches and generate compensations, each query graph is traversed in a bottom-up manner. The attempt is made to bring two boxes at the same 'level' of each query graph into a subsumee-subsumer relationship. Generally, the level of a box in a query graph is the maximum distance of that box to any base table box. If a match between two boxes was successfully established, all combinations of the parent boxes become subject of recording further matches.

To refer to the ongoing example of figure 7, the lowest select boxes for AST1 and AST2 fulfill the necessary conditions. In a first step (figure 8a, (1)), the subsumee box is copied, the predicates are adjusted according to the rules introduced in the preceding section and a quantifier between the current subsumer box and the new compensation box is created (2). Since the subsumee box has an extra incoming quantifier from the region table, a new quantifier from the region table is added to the compensation box (3).

Recording a match at a further level of a query graph, the already existing compensation box is pulled up to the new subsumer box (group by box in the ongoing example of figure 8b) by first copying the box (1) and then swinging the quantifier from the subsumer box below to the current subsumer box (2). Incoming quantifiers from base tables remain unchanged. Thereafter, the current subsumee box is copied (3) and the incoming quantifier is swung to the new compensation box (4).

Finally, the query graph of the subsumee is substituted by the compensation graph built during the generation of the matches. Figure 9 shows the situation after applying the 'Query Stacking' technique to the two sample ASTs from figure 7. The upper half of that new graph corresponds to the newly generated compensation graph, reflecting the 'old' subsumee stacked on top of the subsumer. The region table is joined and the local predicate restricting the query context to 'Europe' is applied in the lowest select box of the compensation.

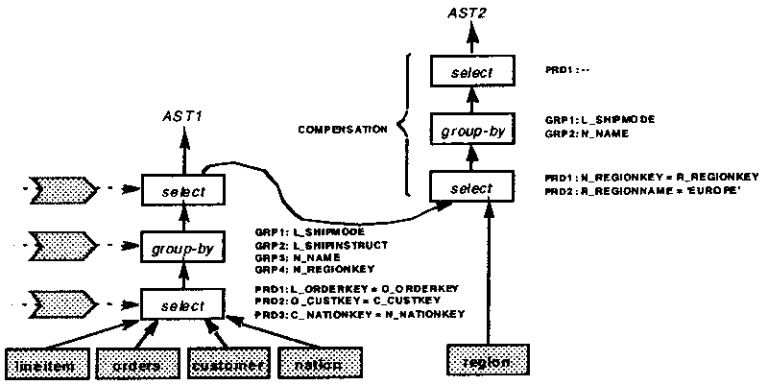


Fig. 9: QGM Graph for Multiple Table Refresh after Query Stacking

Summary

'Query Stacking', where a single query provides some kind of common sub-expression for multiple queries being executed at the same time is a well-known technique from a theoretical point of view. This section introduces the MQO framework of DB2 in the context of this technique. Apparently stacking is not possible if the queries do not follow a strict subsumer-subsumee relationship. For example, if AST2 from figure 7 would not have n_regionkey as a member of the grouping column list, recording a match for the lower select boxes would already fail. The proposed solution, the presentation of which is subject of the following section, tries to generate common sub-expressions systematically. Thus, multiple queries are sharing the results of a common, artificially injected pre-computation step.

4 Query Sharing: Building A Common Subsumer

This section introduces the novel technique of 'Query Sharing'. The general idea of this approach is that, if the query stacking technique is not applicable for two given query graphs, then the system might come up with an artificially constructed common sub-expression, which then can be exploited by both queries. To comply with the already introduced notion of subsumer and subsumee, we refer to such a generated query (sub-)graph as '*common subsumer*'. Again, we demonstrate our approach in the context of the refresh of two summary tables.

4.1 Example of a Common Subsumer

To explain the generation process of a common subsumer, we again refer to the TPC-H/R data schema and concentrate on the optimization of the queries executed to refresh the following two summary tables:

```
SELECT    l_shipdate, l_commitdate,
          SUM(l_quantity) AS sum_qty,
          SUM(l_extendedprice) AS sum_price,
          COUNT(*) AS count_order
FROM      lineitem
GROUP BY  l_shipdate, l_commitdate
```

```
SELECT    l_shipdate, l_receiptdate,
          SUM(l_quantity) AS sum_qty,
          AVG(l_discount) AS avg_disc,
          AVG(l_tax) AS avg_tax,
          COUNT(*) AS count_order
FROM      lineitem
GROUP BY  l_shipdate, l_receiptdate
```

Since neither the select-clauses nor the group-by-expressions are compatible, the execution of the queries could not be optimized by only applying the 'Query Stacking' technique, i.e. exploiting existing common sub-expression. The idea behind the common subsumer, mapped onto this specific example, is to 'massage' the global query graph and inject the following sub-expression. This sub-query can than be shared by both queries as a common sub-expression:

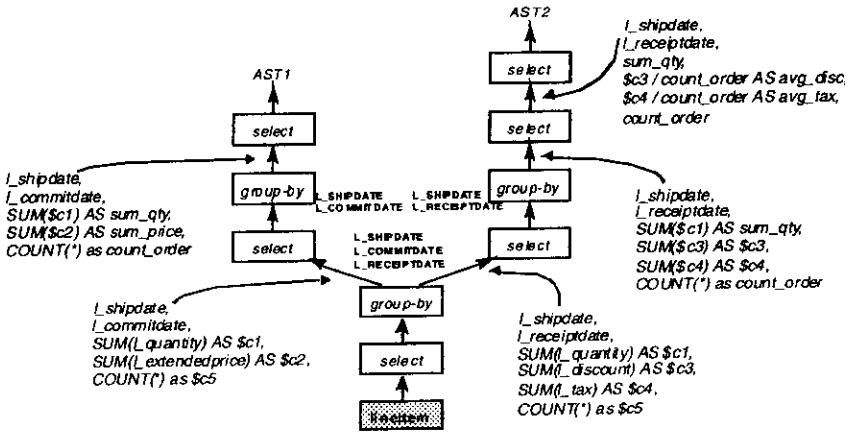


Fig. 10: QGM Graph for the Sample Common Subsumer during Multiple Table Refresh

```
SELECT    l_shipdate, l_commitdate, l_receiptdate,
          SUM(l_quantity),
          SUM(l_extendedprice),
          SUM(l_discount),
          SUM(l_tax),
          COUNT(*)
FROM      lineitem
GROUP BY l_shipdate, l_commitdate, l_receiptdate
```

Figure 10 shows the resulting query graph expressed in QGM terminology.
In general, we have to perform the following „fixes" to make two boxes share a common subsumer:

- union/adjust the select-lists of the participating queries
- delay the application local predicates to compensation or apply a disjunction of local predicates in the common subsumer
- keep extra children and add re-join children
- union/adjust the grouping expressions

We will discuss each „fix" and the implementation extending the query matching technique in the remainder of this section.

4.2 First Level Common Subsumer

The level-oriented production of a common subsumer box consists in general of two separate steps. In a first step, a query box is created and adapted to fulfill the requirements of a common subsumer. Both boxes of the original queries are then matched against that newly constructed common subsumer box. This step generates the necessary compensation boxes, which will replace the original queries after finishing the construction of the whole common subsumer.

Phase 1: Building the Common Subsumer

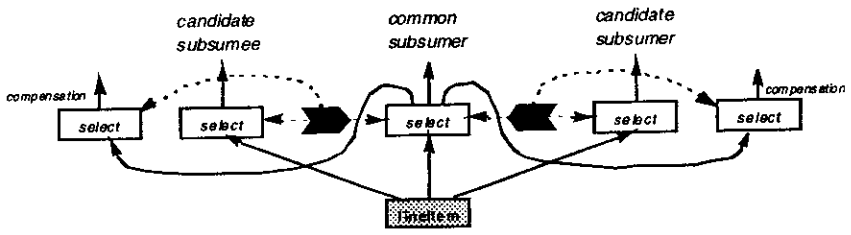


Fig. 12: Recording Matches during Generation of a 1st-Level Common Subsumer

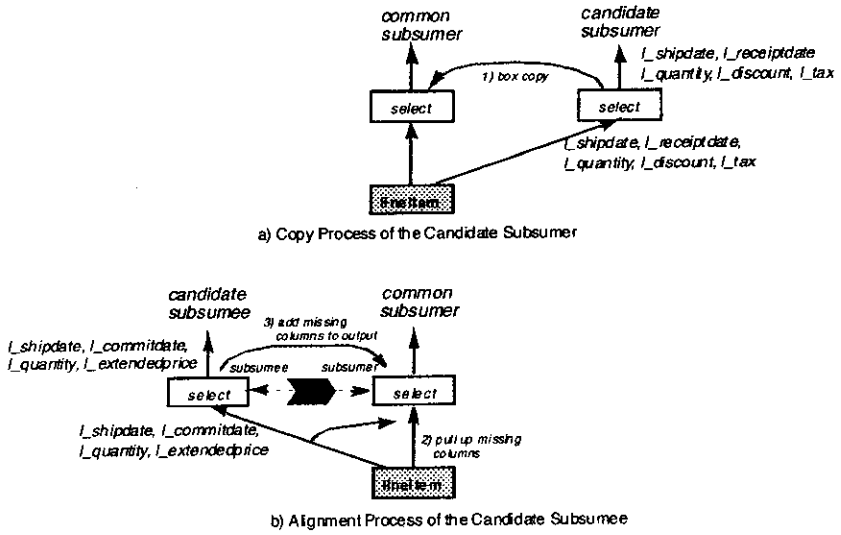


Fig. 11: Generation of a 1st-Level Common Subsumer

After being rejected by the test of stacking compatibility, the common subsumer algorithm is activated with a subsumer candidate and a subsumee candidate box. Instead of building the common subsumer box from scratch, we copy the candidate subsumer box and 'enrich' this box with the missing information that is needed to make it a valid subsumer for the candidate subsumee box, too.

Due to the box copy process, the newly generated common subsumer box inherits the subsumer's input columns from the common base table as well as its output columns** (figure 11a). To make the common subsumer a subsumer for the subsumee, we have to add the missing columns, i.e. those columns which are referenced by the subsumee but not by the subsumer to the input of the common subsumer. Moreover, all columns, which are produced by the subsumee but not by the subsumer have to be added. To accomplish this task, we simulate the recording of a regular subsumee/subsumer match. Each time, the regular matching procedure reports a failure due to violating a matching condition, we try to eliminate this defect by performing the necessary repairs, i.e. adding columns to the input and output of the common subsumer box. Figure 11b illustrates this step for the sample scenario of AST1 and AST2. In this case, the columns l_extendprice and l_commitdate are added to the input as well as to the output of the common subsumer box.

Phase 2: Matching against the Common Subsumer

In the second phase of a common subsumer box generation, we use the regular matching procedures again to record matches between the candidate subsumee and the common subsumer and between the candidate subsumer and the common subsumer. Although there already exists a temporary match between the subsumee and the common subsumer, this match is dropped at the end of the first phase and reestablished to guarantee the correct generation of the associated compensation box. For the same reason, the match between the candidate subsumer and the common subsumer is necessary, despite the original duplication of the two boxes. It is worth mentioning here that the matching against the common subsumer has to be successful, because the common subsumer was exactly designed for that purpose. Figure 12 shows the scenario after finishing the generation of the common subsumer at the first level.

** The 'inheritance' of join and local predicates is discussed in subsection .

4.3 Common Subsumer at Higher Levels

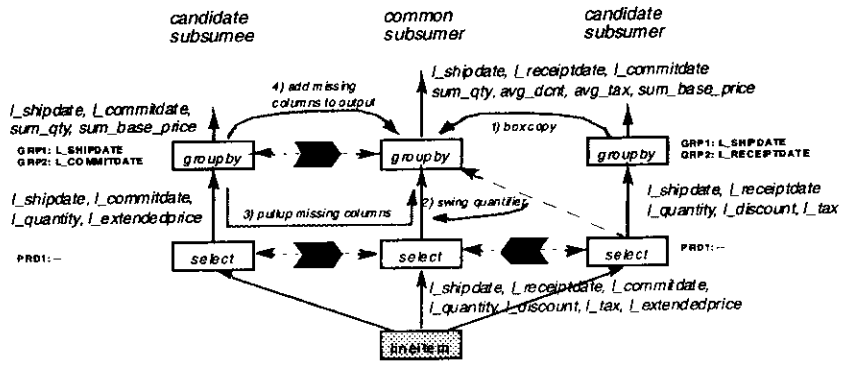


Fig. 13: Generation of a 2nd-Level Common Subsumer

Basically, the generation of a common subsumer at higher levels of a query plan consists of the same phases as a first level common subsumer generation. However, there are some additional steps, which are detailed in this subsection. Figure 13 illustrates the procedure according to the ongoing sample scenario.

After copying of the subsumer box (right group-by box in figure 13: (1)), this box gets its input from the lower box of the candidate subsumer. However to be correct, the copy has to be fed by the common subsumer box of the level below (middle select box in that scenario). To swing the incoming quantifier from the subsumer candidate select box to the common subsumer select box (2) a column mapping between the two boxes is necessary. To generate such a column mapping, we can take advantage of the already recorded match at the lower level: for each column (or expression generating a column) of the subsumee box, we are searching the corresponding column in the subsumer box of that match and record the corresponding column numbers. Since there already exists a match between these two boxes, we can guarantee a successful generation of that column mapping.
In analogy to the first level common subsumer generation, the subsumer copy is enriched by the missing columns and expressions, which are needed to fulfill a subsumer role for the subsumee. Again, missing columns have to be added to the input and to the output of that box. In contrast to the first-level generation, the two boxes do not share a common child box, i.e. a base table. Therefore, the existing match between the subsumee and the common subsumer at the lower level is used to identify the missing columns and add them to the input of the common subsumer box (3). Additionally and contrary to the lower select boxes, it is worth mentioning that in case of group-by boxes not only simple columns, but complete expressions like aggregation functions with the corresponding parameter columns are transferred from the subsumee to the common subsumer (4). In the specific example of figure 13, the columns l_commitdate and SUM(l_extendedprice) are added to the output of the common subsumer.

After finishing the generation of the common subsumer box for that specific level, we again record matches between the candidate subsumee and the common subsumer and between the candidate subsumer and the common subsumer. During that procedure, existing compensations are pulled up to the current level and extended to compensate the differences at the current level.

4.4 Multiple Children Adjustment

To cover multiple children during the generation of a common subsumer, additional steps are necessary after copying the candidate subsumer box. Consider the case when the subsumer contains a table which is not referenced in the candidate subsumee. The inherited quantifier to that child is only kept, if the join corresponds to an existing referential integrity constraint, where the extra child table holds the primary key and the shared table (shared between candidate subsumee and candidate subsumer) holds the foreign key. Otherwise, the overall generation of the common subsumer is aborted.

Figure 15 shows a sample scenario of two queries, where the supplier table is referenced only by the candidate subsumer. Since the associated join realizes a primary/foreign key relationship (ps_suppkey = s_suppkey), the inherited quantifier from the common subsumer box to the candidate subsumer box (and the join predicate - subsection 4.5) is kept without any modifications (1).

```
SELECT    l_shipdate, l_commitdate, s_name,
          SUM(l_quantity) AS sum_qty,
          SUM(ps_availqty) AS sum_availqty,
          COUNT(*) AS count_order
FROM      lineitem, partsupp, supplier
WHERE     l_partkey = ps_partkey
  AND     l_suppkey = ps_suppkey
  AND     ps_suppkey = s_suppkey
GROUP BY l_shipdate, l_commitdate, s_name
```

```
SELECT    l_shipdate, l_commitdate, p_name,
          SUM(l_quantity) AS sum_qty,
          SUM(l_extendedprice) AS sum_price,
          COUNT(*) AS count_order
FROM      lineitem, partsupp, part
WHERE     l_partkey = ps_partkey
  AND     l_suppkey = ps_suppkey
  AND     ps_partkey = p_partkey
GROUP BY l_shipdate, l_commitdate, p_name
```

The same strategy applies if the candidate subsumee references a table, which is not a child of the candidate subsumer. As soon as the join corresponds to a referential integrity constraint, a new quantifier from that child table is added to the common subsumer box (2). All columns, which flow from the child table to the candidate subsumee box are also added to this new quantifier and to the output of the common subsumer box. Since the left query of the ongoing example is grouping by part name (p_name), this query needs a reference to the part table. Thus, a quantifier is added ranging from the part table to the common subsumer box and the p_key and p_name columns are pulled up.

Again, if the join with an extra child does not exhibit the characteristics of 'lossless'-ness, no common subsumer will be generated for that candidate subsumee/candidate subsumer pair. This restrictive strategy is

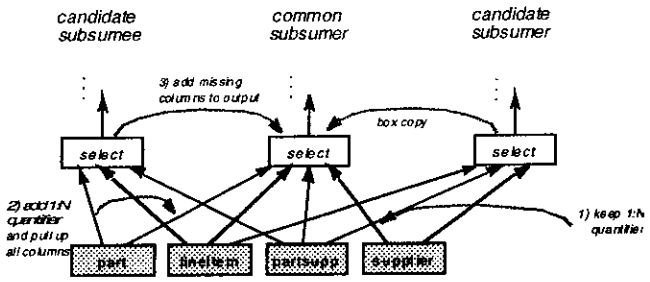


Fig. 14: Common Subsumer in the Presence of Joins

justified by the following two arguments:

- A general join between a shared table and an extra child table may eliminate rows from the shared table, which are needed by the other query. Thus a common subsumer could be built only for the common tables. This strategy however is far too restrictive to serve as a generic sub-expression.
- Since lossless joins following explicitly specified RI constraints are fully supported, handling joins is no limitation in the presence of a star schema, which is considered our main application area for summary tables.

Consider another perspective in this context: why not delay joins with extra children as re-joins to compensation? Of course, this would be technically possible. However, this would also imply that the join columns have to survive the group-by box and thus added to the set of grouping columns. Since those foreign key columns make up the composite primary key in a fact table for example, no desired reduction through aggregation would be achieved by a common subsumer. Therefore, we implemented the strategy of adding extra children connected by an 1:N relationship early as possible so that the join can be performed before the aggregation in the common subsumer.

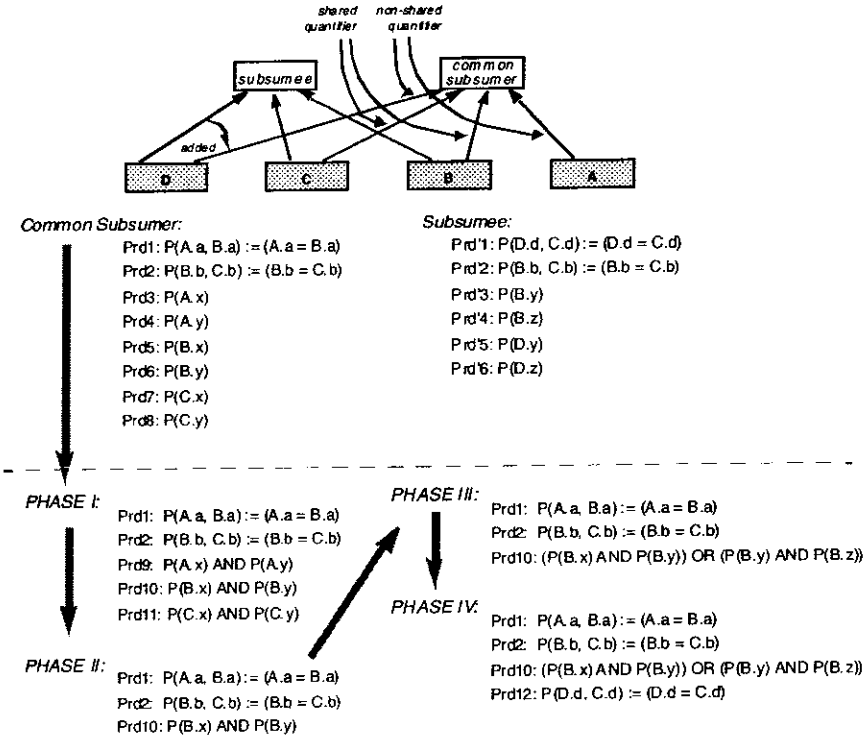


Fig. 15: Sample Scenario for Predicate Adjustment

4.5 Predicate Adjustment

The process of predicate adjustment reflects the most critical part in building a common subsumer for two given queries. The predicate adjustment process must consider the following scenarios:

- common join predicates
- local predicates referring to the same base tables
- unmatched local predicates
- RI-join predicates over not common tables
- local predicates over not common tables

Figure 15 provides a sample scenario covering most of the cases which occur during predicate adjustment. The common subsumer is based on four tables (A, B, C, and D), where table D is local to the subsumee, i.e. a quantifier between D and the common subsumer box was added prior to the predicate adjustment. The current common subsumer inherited all predicates of the candidate subsumer: join predicates between A and B (Prd1) and between B and C (Prd2); local predicates on all its original tables A, B, and C over columns x and y (Prd3 to Prd8). The candidate subsumee shows join predicates between B and C (Prd'2) and between C and D (Prd'1). Additionally, the candidate subsumee exhibits local predicates on the table B and D over columns y and z. With regard to the original configuration of candidate subsumee and candidate subsumer, the quantifiers of the common subsumer from tables A and D are non-shared (because they refer to extra children), whereas quantifiers to the common tables B and C are shared.

The process of predicate adjustment consists of the following four phases. The resulting predicate configuration of the common subsumer box is shown in figure 15 for each single step:

- *Phase I: Predicate Clustering in QGM*
  After query parsing, the predicate of a select box is transferred into conjunctive normal form where each component reflects a simple or disjuncted predicate. The first phase prepares the parse tree for further modification and clusters the single predicates referencing the same set of tables into a single predicate connected by AND.
  In the example, predicates Prd3 to Prd8 are transformed to predicates Prd9, Prd10, and Prd11.
- *Phase II: Predicate Elimination*
  All predicates of the common subsumer with no matching predicate in the subsumee referring to the same set of tables are eliminated; non-local predicates with at least one column coming from a non-shared quantifier (i.e. join predicate to an extra child of the subsumer) are excluded from this rule.
  In the example, Prd9 and Prd11 are eliminated in Phase II (figure 15), because Prd9 is a local predicate referencing an extra child table and Prd5 does not have a 'matching' predicate in the subsumee (like Prd10). Prd1 is spared due to the exception above and Prd2 is spared because it shows a 'matching' predicate in the subsumee (Prd2).
- *Phase III: Predicate Disjunction*
  All predicates of the subsumee with a corresponding predicate referring to the same set of tables are added to the common subsumer under the consideration of predicate subsumption (Psee = predicate of the subsumee, Pcser = predicate of common subsumer):
  - Psee matches exactly Pcser: Do nothing, because the predicates are identical.
  - Pcser subsumes Psee: Do nothing, because Pcser is weaker then Psee.
  - Psee subsumes Pcser:
    Eliminate the predicate Pcser from and add Psee to the common subsumer.
  - no predicate subsumption relationship between Psee and Pcser:
    Extend the common subsumer predicate to (Pcser OR Psee). This case applies to Prd10 and (Prd'3 AND Prd'4) of the subsumee in the ongoing example. Again, before disjunction, the subsumee predicates are combined into a single expression connected by AND.
- *Phase IV: Adding of Extra Child Join Predicates*
  In the last phase, those predicates of the candidate subsumee are added without modification to the common subsumer that are not yet processed within Phase III and show at least one column coming from a non-shared quantifier. In the example, Prd1 of the subsumee is added to the common subsumer in that phase (Prd12).
  To enable the generation of a correct compensation later during the match recording step, the columns of local predicates of the subsumee with no 'matching' predicate in the common subsumer have to be added to the output of the common subsumer.

The proposed predicate adjustment strategy exhibits a relaxation with regard to the 'combination' of local predicates. Consider the following case, where two queries are referring to the same tables R and S with local predicates on x (Q1) and y (Q2), respectively, e.g. Q1: P(R.x) AND P(S.x) and Q2: P(R.y) AND P(S.y)

A disjunction of these predicates results in (P(R.x) AND P(S.x)) OR (P(R.y) AND P(S.y)) or expressed in conjunctive normal from: (P(R.x) OR P(R.y)) AND (P(R.x) OR P(S.y)) AND (P(S.x) OR P(R.y)) AND (P(S.x) OR P(S.y)). Generating this predicate for a common subsumer would imply that no existing index could be exploited due to the disjunctive predicates referring to different tables. For this reason, we relax the predicate for the common subsumer by eliminating the disjunctive terms that reference multiple tables, e.g. the two middle terms in the conjunctive normal form representation above. It is worth mentioning that this predicate relaxation does not have any impact on the query results due to the 'correct' compensations generated during match recording. However this relaxation does have impact on the cardinality of the data stream and size of temporary tables. For example, under the condition of an identical predicate selectivity, the above situation would double the cardinality. However, compared to the 'correct' predicate, the relaxation is not dangerous for the (most expensive) join operation, because the mixed terms could not be pushed down and applied before the join either. Thus, only succeeding operations like sorting for group-by etc. are affected by this predicate relaxation.

4.6 Adjustment of Group-By Columns

Analogous to predicates in select-boxes, the set of grouping columns of the group-by box must be adjusted for the generation of the common subsumer for group-by boxes. This subsection details the necessary steps.

In an obvious first step, missing grouping columns of the candidate subsumee are added to the set of existing grouping columns of the common subsumer. Referring to the example of subsection , the set of grouping columns of the common subsumer is the result of the union on a per column basis:

(l_shipdate, l_commitdate) 'UNION' (l_shipdate, l_receiptdate) = (l_shipdate, l_commitdate, l_receiptdate)

This strategy might turn out very dangerous (section), because the cardinality of the common subsumer data stream can increase significantly. With |gb1| and |gb2| as the cardinalities of the single queries *after* grouping, the cardinality of the common subsumer can increase to |gb1| * |gb2| in the worst case. An alternative for other application areas (esp. AST advisor) might be the union on a per grouping basis, i.e. the use of grouping sets ([SQL99]). This strategy limits the maximal cardinality of the common subsumer to |gb1| + |gb2|. For the ongoing example, the corresponding grouping set expression would be: *grouping sets((l_shipdate, l_commitdate), (l_shipdate, l_receiptdate))*. Since the evaluation of grouping sets is internally resolved to a union of all participating grouping columns, this optimization is not applicable for generating common subsumers in the context of query optimization. However, this strategy provides an enormous potential for further optimization, e.g. consolidating to ROLLUP() or CUBE() expressions in the context of an AST advisor.

In a second step, the list of grouping columns is extended by all columns which are referenced by local predicates within the candidate subsumee and within the candidate subsumer. These columns must survive the group-by, i.e. must become a grouping attribute to enable compensation above this box. For example, extending the definition of AST1 in subsection by WHERE l_shipmode = 'TRUCK' and adding the local predicate l_shipinstruct = 'DELIVER IN PERSON' to the definition of AST2, the common subsumer would result in a group-by query over five columns[††]:

```
SELECT   l_shipdate, l_commitdate, l_receiptdate, l_shipmode, l_shipinstruct,
         SUM(l_quantity),
         SUM(l_extendedprice),
         SUM(l_discount),
         SUM(l_tax),
         COUNT(*)
FROM             lineitem
GROUP BY l_shipdate, l_commitdate, l_receiptdate, l_shipmode, l_shipinstruct
```

Again, extending the list of grouping columns sounds very dangerous with respect to the cardinality of the data stream. However, since we target the data warehousing application area with a star-schema like database schema, restrictions are often made on columns, which are functionally dependent on the original grouping columns, e.g. year = '1999', group by quarter. Adding such columns to the set of grouping columns does therefore not increase the cardinality.

To find all columns involved in local predicates, we traverse all existing compensation boxes, which are based on the child box of the current common subsumer box. Identified columns are then pulled up, i.e. added to the input of the group-by box. Finally, they are added to the list of grouping attributes and to the output of the box.

[††] Note: The local predicate of the candidate subsumer is eliminated from the common subsumer in Phase I and the local predicate of the candidate subsumee is not added to the common subsumer in Phase III of the predicate adjustment process (section)

5 Performance Evaluations

In this section we give and discuss some results of a performance evaluation of the proposed optimization techniques, based on the prototype implementation in DB2. The performance measurements are based on the

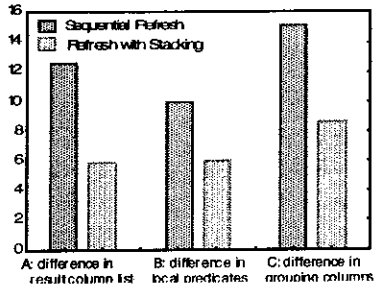


Fig. 16: Stacking for 2 ASTs

TPC/ R scenario with a scaling factor of 0.1 resulting in a 100MByte database. The tests were carried out on an IBM/ RS6000 machine running the IBM DB2 database system developer version 7.1 extended by the implementation of the common subsumer. The following subsections show the runtimes needed for a full refresh of multiple ASTs in sequential mode compared with the runtimes needed when computed simultaneously by using Mass Query Optimization techniques, i.e. either 'Query Stacking' (section) or 'Query Sharing' (section).

5.1 Query Stacking

Figure 16 shows the results of 'Query Stacking' for the refresh of two summary tables. Each query refers to the same set of base tables (lineitem, orders, customer, nation, region as long as not otherwise noted) and exhibits the following differences. In scenario (A), both query bodies are identical, but the queries produce results for different aggregation functions. In scenario (B), one of the two queries exhibits a local predicate. Scenario (C) reflects the situation, where the set of grouping columns of a query is a subset of the grouping columns of the other query. As can be seen from figure 16, the application of the 'Query Stacking' technique results in nearly 50% reduction of the runtime. Notice however that an existing index could be exploited in scenario (B) to speed up the execution of the query with the local predicate in the sequential refresh mode, thus reducing the advantage of the stacked execution.

Consider the situation for three and five summary tables being refreshed simultaneously (figure 17). Again the comparisons between the sequential execution and the execution after rewriting the query plan using 'Query Stacking' are subdivided into scenarios, where the queries differ in the result column list, in the set of local predicates, and in the set of grouping columns.
As expected, excellent speedup is achieved in the case of only a difference in the output list, because the compensation is reduced to a simple projection of the result of the common sub-expression. For more than two participating queries, we introduce a refinement when considering the scenarios with different local predicates. In (B1), one query exhibits a local predicate, whereas in (B2) only one query does not have any local predicates. Although 'Query Stacking' yields a reasonable reduction of the overall runtime, existing physical indexes speed up the queries with a local predicate in the sequential case. It is worth mentioning that the advantage of an existing physical index gets lost when referring to a common sub-expression, if not all participating queries show a restriction on the same table.

Focusing on aggregation, we give the result for the case, where a single query (reflecting the common sub-expression) is grouping over five grouping columns (C). The remaining queries are grouping according to a distinct subset consisting of four different grouping columns of the common sub-expression (4-out-of-5 grouping columns). Again, the 'Query Stacking' approach is far superior with regard to query runtimes.

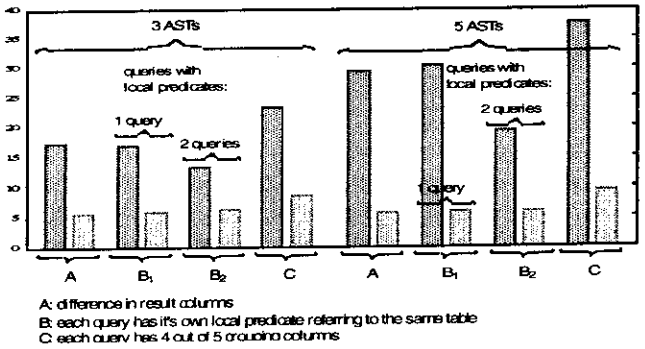


Fig. 17: Refresh of 3/5 ASTs simultaneously using Stacking Optimization

Discussion

As can be seen from these experimental results, exploiting common sub-expressions results in a reasonable reduction of query execution times. Without considering local predicates, a nearly linear reduction to the number of participating queries can be achieved. The existence of local predicates however, may have an impact on the speedup, if their evaluation is delayed to compensation.

5.2 Query Sharing

To turn the focus to the proposed 'Query Sharing' approach using proactive query matching, which artificially injects a common sub-expression into the global query graph, we again consider the situation

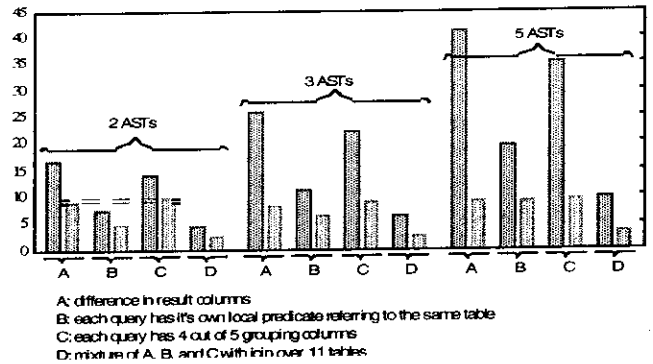


Fig. 18: Using Common Subsumer for Multiple Table Refresh

evaluating two, three, and five queries simultaneously within four different query scenarios. Scenario (A) again reflects identical queries with a difference in the output list. This results in a 'wide' common subsumer and compensations consist of simple projections. Obviously, the speedup again scales nearly linearly with the number of queries. Moreover, it is nice to observe that the overall execution time when applying proactive query matching is independent from the number of participating queries.

In scenario (B), each query exhibits a local predicate referring to the same table, which enables the common sub-expression to generate a corresponding disjunctive expression and use of potentially existing indexes. As can be seen in Figure 18, the generation of a common sub-expression still yields a performance gain compared to the sequential execution.

Scenario (C) addresses different sets of grouping columns. Each query has four out of five different grouping columns. Thus, the common subsumer pre-aggregates according to these five grouping columns. As can be seen especially in the case of two queries, the compensations are more expensive, thus increasing the runtime compared to scenario (A).

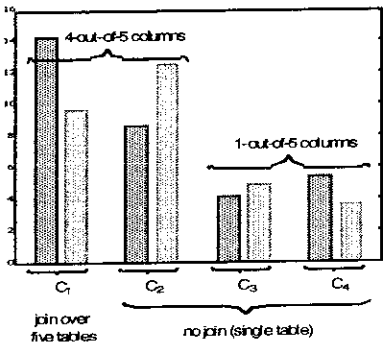
The last scenario (D) simulates a 'typical' application of a common subsumer as a mixture of scenarios (A),

(B), and (C). The queries are defined over 11 tables with restrictions and grouping columns spread over the participating tables. Without going into detail here, we can observe a strong runtime reduction by injecting a common sub-expression.

The Grouping Column Explosion

Referring back to the adjustment of grouping columns when building a common subsumer, we have to recall that the set of grouping columns of a common subsumer results in the union of the grouping column sets of the participating queries. As already mentioned, this may increase the cardinality of the data stream dramatically. An interesting question is, whether computing the sub-expression for non-compatible grouping sets is generally advisable and if not, it is interesting, whether sharing data only for computing the join is most profitable or no sharing at all is the best solution.

Consider therefore the following experimental results building a common sub-expression for four out of five



cardinalities:

	CS	Q1	Q2
C_1	3498	500	875
C_2	60171	60138	60154
C_3	55314	2518	2460
C_4	28	4	7

Fig. 19: Cardinalities of Common Subsumer (2 ASTs)

grouping columns ranging over five tables ((C_1) in figure 19) versus a single table access ((C_2) in figure 19). Surprisingly, the use of a common sub-expression for a single table is more expensive than the sequential execution. If we reduce the grouping columns to 1-out-of-5, we can achieve results in either ways (C_3 and C_4), depending on the intermediate size of the common subsumer. We observe that, if the we have to deal with huge cardinalities, then the sequential execution beats the proactive query matching approach. If we reduce the total size of the common subsumer (scenario (C_4) in figure 19), we again end up with the common subsumer as the better solution.

Discussion

From the presented performance evaluation results we can deduce the following: Injecting a common subsumer box does not always result in a better query plan, which in turn requires a cost-based decision. In the context of DB2, all modifications are performed in the query rewrite phase before calling the optimizer for the rewritten plan. This architecture provides a simple solution in calling the optimizer twice: one call for the unmodified plan and one call for the plan with injected common subsumer. However the fundamental problem of estimating the cardinality of a data stream *after* an aggregation can not be easily solved within a traditional query optimizer. This requires further investigation ([PiCo84], [AAD+96], [RuNT99]). In summary, we have to conclude that the common subsumer technique is highly beneficial for Mass Query Optimization if no extension of the set of grouping columns is required. This limitation can be waived in other application scenarios (like AST advisor), where complex grouping expressions like CUBE(), ROLLUP(), or GROUPING SETS() can be build, thus putting the technique of a common subsumer into the right light.

6 Related Work

The idea of evaluating common parts of a set of queries only once, i.e. the detection of common sub-expressions is nearly as old as query optimization itself (e.g. [Hall74]). First serious attempts to apply Mass Query Optimization techniques in relational database systems can be found in [Jark84] and [Sell88]. These papers try to come up with *general* solutions for the detection of existing common sub-expressions. As shown for example in [SeGh90], the general „Mass Query Processing" problem is NP-hard. While these

articles focus on the theoretical perspective, [AlRa92] gives an overall framework of MQO. The most interesting point in this paper is that they identify sub-problems and explain the dependencies of different optimization directions. Neither aggregation nor the generation of common sub-expression are addressed in these articles.

Newer work related to our application can be found in [RSSB00], [YaKL97], and [ZDNS98]. The work of [RSSB00] targets the specification of greedy algorithms to cost different plans computing common sub-expressions within a single query of across multiple queries. While this approach addresses the algorithmic perspective, we stress the implementation perspective. The goal of the [YaKL97] approach is to identify a set of relations to be materialized so that the total execution plan is minimized when executing all participating queries. Although this approach is related to the proactive query matching strategy under the application perspective of an AST advisor (see section 1), again only existing common sub-expressions are identified and not systematically generated. The only approach targeting the same area can be found in [ZDNS98]. This approach is based on the physical data access level and tries to come up with an optimal access path for multiple queries having similar grouping attributes. Our proactive query matching approach however is based on logical entities (boxes and quantifiers) and applied during query rewrite. This provides the advantage that the optimizer must not be extended and that our approach can be seamlessly integrated on top of a traditional optimizer. Furthermore, applying the optimization during rewriting enables us to construct complex compensation graphs, which would be a very tough job for an optimizer.

7 Summary and Future Work

The general idea of the proactive query matching technique is to systematically generate a common sub-expression for a set of similar queries. This paper gives an introduction of the existing matching technology in IBM DB2/UDB and details the necessary extensions for proactive query matching. The overall strategy is as simple as effective: try to establish a match and - if not successful - apply specific repair actions to make the match happen. Thus, using the regular matching technique we are able to identify common sub-expression, whereas applying the proactive query matching technique, we are able to construct specially designed common sub-expressions, which did not exist in the original query graphs. Both strategies are applied to the application of simultaneously re-computing multiple materialized views, resulting in the 'Query Stacking' and 'Query Sharing' optimization technique.

As sketched in the performance evaluation section, the generation of a common sub-expression is not generally advisable, but dependent on the size of the common subsumer. Therefore further work is investigating the question how to estimate the cardinality of a data stream after performing the aggregation. In summary, the proposed proactive query matching technology provides a sound basis for further investigations in the context of Mass Query Optimization. This area will definitely gain more and more attention in the near future.

References

AAD+96 Agrawal, S.; Agrawal, R.; Deshpande, P. M.; Gupta, A.; Naughton, J.F.; Ramakrishnan, R.; Sarawagi, S.: On the Computation of Multidimensional Aggregates. In: *22th International Conference on Very Large Data Bases* (VLDB'96, Bombay, India, Sept. 3-6), 1996, pp. 506-521

AdLi80 Adiba, M.E.; Lindsay, B.G.: Database Snapshots. In: *6th International Conference on Very Large Data Bases* (VLDB'80, Montreal, Canada, Oct. 1-3), 1980, pp. 86-91

AlRa92 Alsabbagh, J.R.; Raghavan, V.V.: A Framework for Multiple Query Optimization. In: *2nd International Workshop on Research Issues on Data Engineering: Transaction and Query Processing* (RIDE'92, Tempe (AZ), USA, Feb. 2-3), 1992, pp. 157-162

ChIo98 Chan, C.-Y.; Ioannidis, Y.E.: Bitmap Index Design and Evaluation. In: *27th International Conference on Management of Data* (SIGMOD'98, Seattle (WA), USA, June 2-4), 1998, pp. 355-366

ChMc89 Chen, M.C.; McNamee, L.P.: On the Data Model and Access Method of Summary Data

Management. In: *IEEE Transactions on Knowledge and Data Engineering 1(1989)4*, pp. 519-529

GBLP96 Gray, J.; Bosworth, A.; Layman, A.; Pirahesh, H.: Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. In: *12th International Conference on Data Engineering* (ICDE'96, New Orleans (LA), USA, Feb. 26- Mar. 1), 1996, pp. 152-159

GuMu99 Gupta, A.; Mumick, I.S.: Materialized Views : Techniques, Implementations, and Applications, 1999

Hall74 Hall, P.V.: Common subexpression identification in general algebraic systems. *Technical Report UKSC 0060, IBM United Kingdom Scientific Centre, Nov, 1974.*

HaRU96 Harinarayan, V.; Rajaraman, A.; Ullman, J.D.: Implementing Data Cubes Efficiently. In: *25th International Conference on Management of Data* (SIGMOD'96, Montreal, Canada, June 4-6), 1996

Jark84 Jarke, M.: Common subexpression isolation in multiple query optimization. In: *Query Processing in Database Systems*, pp. 191-205, 1984

LSPC00 Lehner, W.; Sidle, R.; Pirahesh, H.; Cochrane, B.: Maintenance of Automatic Summary Tables in IBM DB2/UDB. In: *29th International Conference on Management of Data* (SIGMOD'2000, Dallas (TX), USA, May 14-19), 2000

ONei87 O'Neil, P.: Model 204: Architecture and Performance. In: Gawlick, D.; Haynie, M.; Reuter, A. (Editors): *High Performance Transaction Systems. Lecture Notes in Computer Science 359*, 1987

PiCo84 Piatesky-Shapiro, G.; Connell, C.: Accurate Estimation of the Number of Tuples Satisfying a Condition. In: *Annual Meeting on Management of Data* (SIGMOD'84, Boston (MA), June, 18-21), 1984, pp. 256-276

PiLH97 Pirahesh, H.; Leung, C.; Hasan, W.: A Rule Engine for Query Transformation in Starburst and IBM DB2 C/S DBMS. In: *13th International Conference on Data Engineering* (ICDE'97, Birmingham, U.K., April 7-11), 1997, pp. 391-400

RSSB00 Roy, P.; Seshadri, S.; Sudarshan, S.; Bhobe, S.: Efficient Algorithms for Multi Query Optimization. In: *29th International Conference on Management of Data* (SIGMOD'2000, Dallas (TX), USA, May 14-19), 2000

RuNT99 Runapongsa, K.; Nadeau, T.P.; Teorey, T.: Storage Estimation for Multidimensional Aggregates in OLAP. In: *CASCON'99 Conference* (Toronto, CA, Nov. 8-11), 1999, pp. 40-54

SeGh90 Sellis, T.; Ghosh, S.: On the Multiple-Query Optimization Problem. In: *IEEE Transactions on Knowledge and Data Engineering 2(1990)2*, pp. 262-266

Sell88 Sellis, T.K.: Multiple Query Optimization. In: *ACM Transactions on Database System, 13(1988)1*, pp. 51

SQL99 N.N.: *ISO/IEC 9075:1999* Information technology – Database languages -- SQL, 2000

TPC99 N.N.: *TPC-R Benchmark Specification Rev. 1.0.1.* Transaction Processing Performance Council, 1999 (http://www.tpc.org/rspec.html)

YaKL97 Yang, J.; Karlapalem, K.; Li, Q.: Algorithms for Materialized View Design in Data Warehousing Environment. In: *23rd International Conference on Very Large Data Bases* (VLDB'97, Athens, Greece, Aug. 25-29), 1997, pp. 136-145

Vald87 Valduriez, P.: Join Indices. In: *ACM Transactions on Database Systems* 12(1987)2, pp. 218-246

ZDNS98 Zhao, Y.; Deshpande, P.M.; Naughton, J.F.; Shukla, A.: Simultaneous Optimization and Evaluation of Multiple Dimensional Queries. In: *27th International Conference on Management of Data* (SIGMOD'98, Seattle (WA), USA, June 2-4), 1998, pp. 271-282

ZCP+00 Zaharioudakis, M.; Cochrane, R.; Pirahesh, H.; Lapis, G.; Urata, M.: Answering Complex SQL Queries Using Summary Tables. In: *29th International Conference on Management of Data* (SIGMOD'2000, Dallas (TX), USA, May 14-19), 2000