Wolfgang Lehner, Bobbie Cochrane, Hamid Pirahesh, Markos Zaharioudakis

fAST Refresh using Mass Query Optimization

Erstveröffentlichung in / First published in:

Proceedings 17th International Conference on Data Engineering. Heidelberg, 02.-06.04.2001. IEEE, S. 391-398. ISBN 0-7695-1001-9

DOI: https://doi.org/10.1109/ICDE.2001.914852

Final edited form was published in "Proceedings 17th International Conference on Data Engineering", Heidelberg, 2001. ISBN 0-7695-1001-9.
https://doi.org/10.1109/ICDE.2001.914852

fAST Refresh using Mass Query Optimization

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

Automatic Summary Tables (ASTs), more commonly known as materialized views, are widely used to enhance query performance, particularly for aggregate queries. Such queries access a huge number of rows to retrieve aggregated summary data while performing multiple joins in the context of a typical data warehouse star schema. To keep ASTs consistent with their underlying base data, the ASTs are either immediately synchronized or fully recomputed. This paper proposes an optimization strategy for simultaneously refreshing multiple ASTs, thus avoiding multiple scans of a large fact table (one pass for AST computation). A query stacking strategy detects common sub-expressions using the available query matching technology of DB2. Since exact common sub-expressions are rare, the novel query sharing approach systematically generates 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/UWO 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 were developed in this context over the last few years. At the same time, many old ideas have come to a rejuvenation and were adapted to the warehouse application scenario. We follow the same approach and pick up the idea of Mass Query Optimization (MQO: [6], [11]) to simultaneously compute multiple ASTs. The approach, performing an initial population or a full refresh of a set of ASTs ([4]) within a single pass hides a tremendous optimization potential.

In this paper we outline the currently existing MQO framework of the IBM DB2 UDB/UWO database system and propose an extension to this framework to simultaneously compute the set of underlying queries of registered ASTs. The general idea of our proposed optimization technique is to massage a query graph in two ways: if two queries share a common subexpression, we may utilize it. However, if two queries are similar but do not exhibit a common subexpression relationship, we design, build and inject a minimal subgraph, acting as a common subexpression for these queries.

Contribution of the Paper

Although not restricted to this area, we demonstrate the feasibility of our optimization technique in the context of simultaneously refreshing multiple ASTs by optimizing the corresponding view definition queries. In detail, the paper provides contribution in the following areas:

- The Query Graph Model and a generic MQO framework for query matching based on [16] is described.
- Since the proposed optimization technique does not only recognize existing common sub-expressions but artificially generates generic common sub-expressions, it applies to a much wider class of queries than prior work.
- The proposed techniques are independent of specific query patterns, implying that ASTs over (nearly) arbitrary queries (nested, correlated, ...) may be subject of optimization.
- The presentation of our approach and all performance evaluations are based on the TPC-H/R database schema ([13]).
- We give some performance figures based on a prototype implementation within the IBM DB2 UDB/UWO database system.
- 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.

Structure of the Paper

The following section sketches the query matching framework of the IBM DB2 UDB/UWO database system. Section 3 explains the application of these matching techniques and introduces the query stacking optimization approach. Section 4 focuses on the novel strategy of query sharing, which is evaluated by means of experiments in section 5. Section 6 discusses related work in the area of simultaneously optimizing multiple queries. The paper closes with a summary in section 7.

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

Final edited form was published in "Proceedings 17th International Conference on Data Engineering", Heidelberg, 2001. ISBN 0-7695-1001-9.
https://doi.org/10.1109/ICDE.2001.914852

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, [8]) and summarizes necessary conditions to successfully establish a match relationship between two queries. For a detailed description of the DB2 matching technique in the context of transparently routing user queries to existing summary tables, we refer to [16].

2.1. Example

To illustrate the proposed techniques, consider the following query computing 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 ([13]), where the table lineitem holds transactional data, which are evaluated according to the three 'dimensions' of orders, parts, and suppliers.

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.

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 is depicted in figure 1. In general, the QGM data structures consist of boxes which are connected by directed arcs (quantifiers). Boxes represent data sources, e.g. base tables or table functions, and logical operators. *Select* boxes stand for applying local predicates and/or performing join operations. Group-by boxes aggregate along grouping columns, complex grouping expressions using CUBE(), ROLLUP(), GROUPING SETS() expressions, or a combination of both ([3]). Quantifiers denote the data streams flowing from one box to another one.

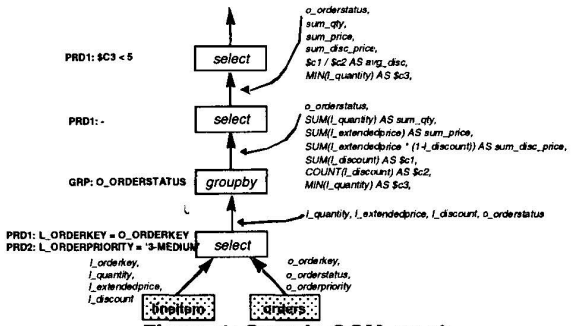

Figure 1: Sample QGM graph

Each box may have multiple outgoing quantifiers (consumers), i.e. a specific box may be a child of 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.*

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 may 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 'match' relationship. In the context of a match, 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 2). 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 for successfully substituting a subsumee by the corresponding subsumer.

- *General Matching Conditions*

Recording a match between two QGM boxes requires

Final edited form was published in "Proceedings 17th International Conference on Data Engineering", Heidelberg, 2001. ISBN 0-7695-1001-9.
https://doi.org/10.1109/ICDE.2001.914852

that both candidate boxes exhibit the same box type and that base table boxes refer to the same data source. Moreover 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.

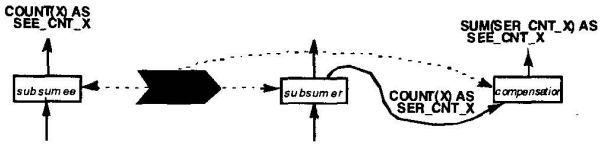

Figure 2: Example of a Match and a Compensation

- *Match Conditions for Local Predicates*
  To record a match 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. If the local predicates of a subsumer and subsumee are identical, then no compensation predicate is necessary.
- *Matching Conditions for Joins*
  Join predicates of two boxes must in general match exactly and the two candidate boxes must reference the same set of children. However, the existence of a lossless 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.
  - *Rejoin of Additional Children*
    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. This of course requires that the necessary join column survives the subsumer.
- *Match Conditions for Grouping Columns*
  The grouping expression of the subsumee must be derivable from the grouping expression of the subsumer. In general both boxes may exhibit arbitrary complex grouping expressions. For the remainder of the paper we restrict the presentation to simple grouping columns where the subsumee grouping columns must be a subset of the subsumer grouping columns[‡].

[‡]Note that additionally the set of group-by columns of the subsumer has to encompass all local predicate columns of the subsumee.

2.4. 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:

```
CREATE SUMMARY TABLE 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 TABLE 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 ) ...
```

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.

3. 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 stacking of queries.

Final edited form was published in "Proceedings 17th International Conference on Data Engineering", Heidelberg, 2001. ISBN 0-7695-1001-9.
https://doi.org/10.1109/ICDE.2001.914852

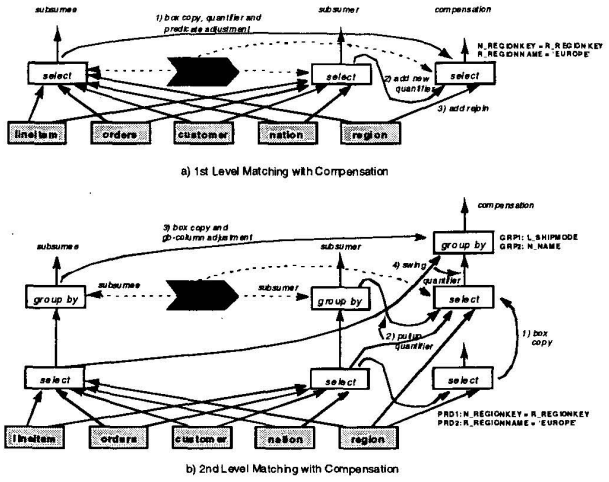

Figure 3: Matching with Compensation

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. 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, the lowest select boxes for AST1 and AST2 fulfill the necessary conditions. In a first step (figure 3a, (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).

For 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 3b) 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 4 shows the situation after applying the query stacking technique to the two sample ASTs. 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.

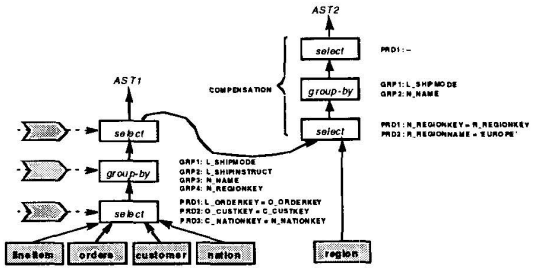

Figure 4: QGM Graph after Query Stacking

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. Apparently stacking is not possible if the queries do not follow a strict subsumer-subsumee relationship. For example, if AST2 would not have n_regionkey as a member of the grouping column list, recording a match between the lower select boxes would already fail.

4. Building A Common Subsumer

The general idea of query sharing 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,

Final edited form was published in "Proceedings 17th International Conference on Data Engineering", Heidelberg, 2001. ISBN 0-7695-1001-9.
https://doi.org/10.1109/ICDE.2001.914852

mapped onto this specific example, is to 'massage' the global query graph and inject an artifical minimal common sub-expression. 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

4.2. First Level Common Subsumer

The level-oriented production of a common subsumer box consists in two separate steps of building and matching against the common subsumer.

Phase 1: Building the Common Subsumer

The general procedure is to copy the candidate subsumer box and 'enrich' it with the missing information that is needed to make it a valid subsumer for the candidate subsumee box.

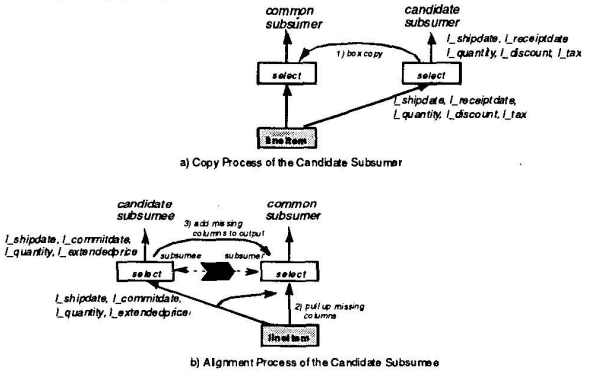

Figure 5: Generation of a 1st Level Common Subsumer

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 5a). 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 bb illustrates this step, where 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 6 shows the scenario after finishing the generation of the common subsumer at the first level.

4.3. Common Subsumer at Higher Levels

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 outlined in this subsection. Figure 6 illustrates the procedure according to the ongoing sample scenario.

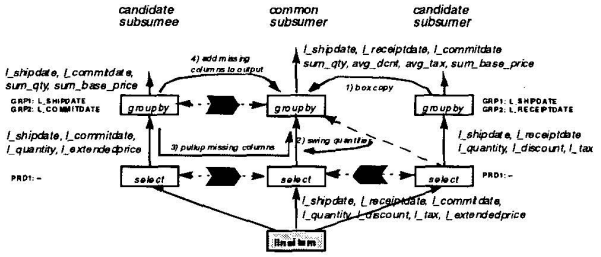

Figure 6: Generation of a 2nd Level Common Subsumer

After copying of the subsumer box (right group-by box in figure 6: (1)), this box is fed 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.

**The 'inheritance' of join and local predicates is discussed in subsection 4.6 .

Final edited form was published in "Proceedings 17th International Conference on Data Engineering", Heidelberg, 2001. ISBN 0-7695-1001-9.
https://doi.org/10.1109/ICDE.2001.914852

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 6, 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 7 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).

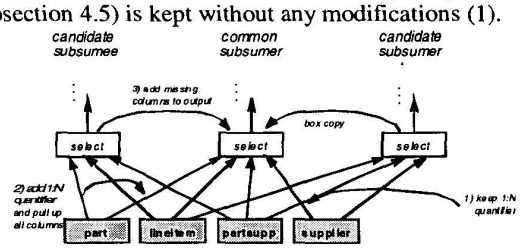

Figure 7: Common Subsumer in the Presence of Joins

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), it 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.

Note however that, 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.

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

After transforming the predicates into conjunctive normal form, the process of predicate adjustment consists of the following three phases:

- *Phase I: 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.
- *Phase II: 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 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).

Final edited form was published in "Proceedings 17th International Conference on Data Engineering", Heidelberg, 2001. ISBN 0-7695-1001-9.
https://doi.org/10.1109/ICDE.2001.914852

- *Phase III: 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 II and show at least one column coming from a non-shared quantifier. 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 in the common subsumer 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. 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.

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. This strategy however might turn out very dangerous, 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 might be the union on a per grouping basis, i.e. the use of grouping sets ([7], [12]), limiting the maximal cardinality of the common subsumer to |gb1| + |gb2|.

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 (like year) to the set of grouping columns does therefore not increase the cardinality.

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/v7 using a 100MByte TPC/R scenario. The experiments show the runtimes needed for a full refresh of multiple ASTs in sequential mode compared to the runtimes needed when computed simultaneously by using the proposed common subsumer techniques. Moreover, each query refers to the same set of base tables (lineitem, orders, customer, nation, region as long as not otherwise noted). Different scenarios are used to model certain behavior.

Scenario (A) in figure 8 shows the runtimes for 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 with CS is independent from the number of participating queries.

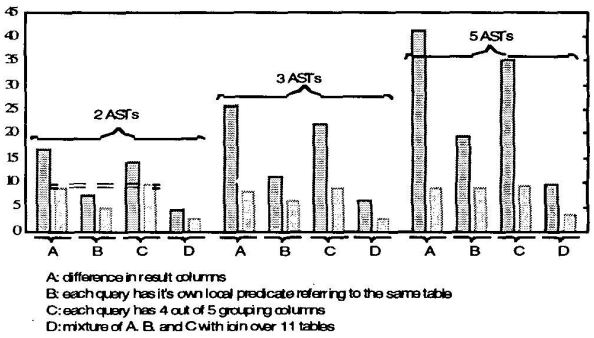

Figure 8: Common Subsumer for Multiple AST Refresh

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 8, 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

Final edited form was published in "Proceedings 17th International Conference on Data Engineering", Heidelberg, 2001. ISBN 0-7695-1001-9.
https://doi.org/10.1109/ICDE.2001.914852

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 subexpression.

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. [5]). First serious attempts to apply Mass Query Optimization techniques in relational database systems can be found in [6] and [11]. These papers try to come up with *general* solutions for the detection of existing common sub-expressions. As shown for example in [10], the general „Mass Query Processing" problem is NP-hard. While these articles focus on the theoretical perspective, [2] gives an overall framework of MQO. The most interesting point in this paper is that they identify subproblems 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 [9], [1], [14], and [15]. The work of [9] targets the specification of greedy algorithms to cost different plans computing common subexpressions within a single query of across multiple queries. While this approach addresses the algorithmic perspective, we stress the implementation perspective. The only approach targeting the same area can be found in [14]. 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

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/UWO and details the necessary extensions to design, build and inject an artificial common subexpression into a global query graph for multiple local queries. 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 not only able to identify common sub-expression to construct specially designed common sub-expressions, which did not exist in the original query graphs. Both strategies are prototypical implemented and evaluated in the context of simultaneously refreshing multiple ASTs but are subject of optimization in other application scenarios where the focus are complex grouping expression like CUBE(), ROLLUP(), or GROUPING SETS().

References

[1] Agrawal, S.; Agrawal, R.; Deshpande, P. M.; Gupta, A.; Naughton, J.F.; Ramakrishnan, R.; Sarawagi, S.: On the Computation of Multidimensional Aggregates. In: *VLDB'96*

[2] Alsabbagh, J.R.; Raghavan, V.V.: A Framework for Multiple Query Optimization. In: *RIDE'92*

[3] Gray, J.; Bosworth, A.; Layman, A.; Pirahesh, H.: Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. In: *ICDE'96*

[4] Gupta, A.; Mumick, I.S.: *Materialized Views: Techniques, Implementations, and Applications*, MIT Press, 1999

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

[6] Jarke, M.: Common subexpression isolation in multiple query optimization. In: *Query Processing in Database Systems*, 1984

[7] Lehner, W.; Sidle, R.; Pirahesh, H.; Cochrane, B.: Maintenance of Automatic Summary Tables in IBM DB2/UDB. In: *SIGMOD'2000*

[8] Pirahesh, H.; Leung, C.; Hasan, W.: A Rule Engine for Query Transformation in Starburst and IBM DB2 C/S DBMS. In: *ICDE'97*

[9] Roy, P.; Seshadri, S.; Sudarshan, S.; Bhobe, S.: Efficient Algorithms for Multi Query Optimization. In: *SIGMOD'2000*

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

[11] Sellis, T.K.: Multiple Query Optimization. In: *ACM Transactions on Database System, 13(1988)1*

[12] N.N.: *ISO/IEC 9075:1999 Information technology -- Database languages -- SQL*, 2000

[13] N.N.: *TPC-R Benchmark Specification Rev. 1.0.1.* Transaction Processing Performance Council, 1999

[14] Yang, J.; Karlapalem, K.; Li, Q.: Algorithms for Materialized View Design in Data Warehousing Environment. In: *VLDB'97*

[15] Zhao, Y.; Deshpande, P.M.; Naughton, J.F.; Shukla, A.: Simultaneous Optimization and Evaluation of Multiple Dimensional Queries. In: *SIGMOD'98*

[16] Zaharioudakis, M.; Cochrane, R.; Pirahesh, H.; Lapis, G.; Urata, M.: Answering Complex SQL Queries Using Summary Tables. In: *SIGMOD'2000*