W. Lehner, W. Hümmer, L. Schlesinger, A. Bauer

On the problem of generating common predecessors

Erstveröffentlichung in / First published in:

CIKM02: Eleventh ACM International Conference on Information and Knowledge Management, McLean 08.11.2002. ACM Digital Library, S. 43–48. ISBN 978-1-58113-590-9

DOI: https://doi.org/10.1145/583890.583897

Final edited form was published in "CIKM02: Eleventh ACM International Conference on Information and Knowledge Management. McLean 2002", S. 43–48,
ISBN 978-1-58113-590-9
https://doi.org/10.1145/583890.583897

On the Problem of Generating Common Predecessors

W. Lehner, W. Hümmer, L. Schlesinger, A. Bauer
Department of Database Systems, University of Erlangen-Nuremberg, Martensstr. 3, 91058 Erlangen, Germany
{lehner ,huemmer ,schlesinger, bauer}@informatik.uni-erlangen.de

ABSTRACT

Using common subexpressions to speed up a set of queries is a well known and long studied problem. However, due to the isolation requirement, operating a database in the classic transactional way does not offer many applications to exploit the benefits of simultaneously computing a set of queries. In the opposite, many applications can be identified in the context of data warehousing, e.g. optimizing the incremental maintenance process of multiple dependent materialized views or the generation of application specific data marts. In the paper we discuss the problem whether it is always advisable to generate the most complete common predecessor for a given set of queries or to restrict a predecessor to a subset of all possible base tables. As we will see, this question cannot be answered without having knowledge about the cardinality of queries after aggregation. However, if we can rely on this information, we can come up with an optimal predecessor for a common set of queries.

1 INTRODUCTION

Data warehousing is considered so differently with regard to transactionally operating a database system that many new aspects have been discovered (e.g. the data cube: [5], [1]) and applied to the application area. One example is the simultaneous execution of a set of queries. Due to the ACID ([8]) properties of transactional queries, the mechanism of multiple query optimization (MQO; [13], ..., [16], ...) was and is still not appropriately acknowledged in research and commercial development. In the opposite, as we will see in the following examples, the data warehousing application area provides a tremendous potential for this technique (e.g. [7], [11]):

- *Decomposing OLAP Queries:* Usually an analyst is using a specific OLAP tool to formulate queries within the multidimensional world. These multidimensional queries are then sent to an OLAP server transforming those queries into a sequence of (depending on the tool: very simple) SQL queries which might be executed in parallel as far as no dependencies exist between the single SQL queries).
- *Refreshing Materialized Views:* Using materialized views in a data warehouse system usually boosts performance in the order of magnitude, because the raw data access is avoided in favor of just referring to precomputed data ([4], [14], [11]). The downside however is that materialized views have to be synchronized with changes to the original base tables. Depending on the proposed maintenance strategy, we may take advantage of optimizing a set of single queries ([9]):
  - *Complete Refresh:* The most obvious situation in dealing with a set of queries is to fully recompute materialized views by executing the defining queries. These queries reflect perfect candidates for common optimization so that scanning the typically very huge fact table could be reduced to a single pass instead of accessing it for every query.
  - *Incremental Refresh:* Only rows affected by an update operation of the base tables are propagated to the materialized views. Deltas based on these rows are computed according to the view definition, so that optimizing these delta by maintaining multiple dependent materialized views sounds extremely attractive to speed up the update process (including the incremental view maintenance) of the underlying base tables.

To put it into a nutshell, optimizing a set of queries sounds beneficial in the context of query processing in data warehousing. Although the general question of multiple query optimization was discussed in early research papers, data warehousing implies new facets for this problem.

Problem Description

Instead of addressing the most general problem of multiple query optimization, we concentrate on the following subset. Given a set of queries $Q_1,...,Q_n$ we try to come up with a proper solution to the question: *Is there a common and artificially build predecessor query which might be computed first and then exploited to speed up many queries referring to the result of this query?* Moreover we try to answer the question more accurately so far as we are investigating the problem into two more directions: Is such a common predecessor (CP) unique – or the other way round – what is the shape of the optimal common predecessor? Furthermore we consider the problem of partitioning, i.e. are there many common predecessors which might be used in performing work that is beneficial for many queries of the given query set.

Structure of the Paper

The following section sketches some work done in this area and concludes that no work is known dealing with joins and aggregation to generate a common predecessor to subsequently speed up queries exploiting this CP. Section 3 introduces the necessary concept of compensation queries and outlines some rules which must be followed during the construction process of a candidate CP. Section 4 proposes the join lattice data structure to organize all possible CPs for a given schema and query scenario. Thereafter, this join lattice will be extended in the case of a snowflake and an underlying galaxy database schema. Section 5 finally details the CP selection algorithm in combination with the applied cost model. The description is illustrated by examples from the TPC-H scenario demonstrating that the naive way of building a CP does not always yield the optimal solution. The paper closes with a summary in section 6.

https://doi.org/10.1145/583890.583897

Final edited form was published in "CIKM02: Eleventh ACM International Conference on Information and Knowledge Management. McLean 2002", S. 43–48,
ISBN 978-1-58113-590-9
https://doi.org/10.1145/583890.583897

2 COMPENSATIONS IN THE PRESENCE OF COMMON PREDECESSORS

This section introduces the technique of compensation and gives rules to generate a common predecessors (CP). Before diving into detail however, we outline the sample scenario, which will be used throughout the paper.

Sample Scenario

To illustrate the proposed techniques of building CPs based on the mechanism of compensation queries sketched below we refer to the TPC-H database schema (*http://www.tpc.org*). In this schema, a 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, so that the database engine is able to take advantage of this information.

Basic Principle of Compensation

The idea of compensation was introduced in the context of routing incoming user queries transparently to existing materialized views or summary tables ([11], [3]). Given Q as the user query and Q' as the query describing the materialized view, it is possible to derive Q from Q' even if Q' is a common subexpression and produces therefore 'more' and 'weaker' information than needed by Q. Thus Q^c is usually generated as a compensation query to perform the remaining work, e.g. applying missing predicates and performing aggregations at higher levels to achieve query equivalence, i.e.

$$Q \equiv Q^c (Q')$$

It is worth mentioning that the query Q' must exhibit a finer granularity schema (i.e. *more* grouping columns) and all columns needed to perform additional predicates in Q must survive the aggregation operation of Q' (i.e. they must be a member of the grouping list of Q'). For example the following query can be derived from the materialized view defined by query Q':

Query Q:

```
SELECT L_SHIPMODE, N_NAME,
       SUM(L_QUANTITY) AS SUM_QTY,
  FROM LINEITEM, ORDERS, CUSTOMER, NATION
 WHERE L_ORDERKEY = O_ORDERKEY
   AND O_CUSTKEY = C_CUSTKEY
   AND C_NATIONKEY = N_NATIONKEY
   AND L_SHIPINSTRUCT IN
         ('DELIVER IN PERSON','TAKE BACK RETURN')
GROUP BY L_SHIPMODE, N_NAME
```

Materialized View defined by Query Q':

```
SELECT L_SHIPMODE, L_SHIPINSTRUCT, C_MKTSEGMENT,
       C_REGIONKEY, SUM(L_QUANTITY) AS SUM_QTY,
  FROM LINEITEM, ORDERS, CUSTOMER
 WHERE L_ORDERKEY = O_ORDERKEY
   AND O_CUSTKEY = C_CUSTKEY
GROUP BY L_SHIPMODE, L_SHIPINSTRUCT,
         C_MKTSEGMENT, C_REGIONKEY
```

The compensation query to produce the correct result with regard to Q based on Q' is the following expression. Note that the l_shipinstruct column of the local predicate in Q is a member of the grouping list of Q' and the missing information regarding the nation name is generated by rejoining Q' with the corresponding nation table on c_nationkey:

```
SELECT L_SHIPMODE, N_NAME, SUM(SUM_QTY) AS SUM_QTY,
  FROM Q', NATION
 WHERE C_NATIONKEY = N_NATIONKEY
   AND L_SHIPINSTRUCT IN
         ('DELIVER IN PERSON', 'TAKE BACK RETURN')
GROUP BY L_SHIPMODE, N_NAME
```

Rules to Generate a CP

Before diving into the discussion regarding the generation of a CP for a given set of queries, we may introduce the following notation: $g(a_1,...,a_n)$ denotes the set of grouping columns a_i ($1 \le i \le n$) of a given relational expression. $g[R](a_1,...,a_n)$ restricts the set of all grouping columns of an expression to those from table R. Analogously, $p(a_1,...,a_n)$ (or $p[R](a_1,...,a_n)$) refers to the set of columns forming the local predicate of the relational expression (or the table R). At last, $j(a_1,...,a_n)$ is used to represent the set of columns involved in specifying a join predicate. Obviously the set does not imply a partitioning scheme of the set of columns but may exhibit overlaps, i.e. a columns may be used in a local predicate and in a grouping expression.

Given a set of relational expressions, we may now provide some rules to generate a CP:

- *Align Grouping Combinations:* The basic rule to align the grouping combinations is that the set of grouping columns is the result of the union of the grouping columns of the participating queries. For two queries Q_1 with g(A,B) and Q_2 with g(B,C) respectively, we get g(A,B,C)[1] as the set of grouping columns for the CP to completely feed Q_1 and Q_2.
- *Align Local Predicates:* Since local predicates are 'local' to each participating query, they have to be deferred to the compensation. As already sketched in the preceding example, we also have to add the set of all local predicate columns p() of each query to the set of grouping attributes. We may additionally optimize this step, if all participating queries have a local predicate using the same column. In this case, we may push down the predicate evaluation into the computation of the CP so that there is no longer a need to consider this column and adding it to the set of grouping attributes.
  For example, having Q_1 with p(X,Y) and Q_2 with p(Y,Z) we may end up with p(Y) and g(...,X,Z) for the CP.
- *Align Join Predicates:* Similarly to the process of aligning local predicates, we have to add the join columns of a query to the set of grouping columns if not all tables of that query are used to build the CP. As we will see in the following section, the optimal CP does not always contain all possible tables of a multiple query scenario. Thus, if it is determined that a table R of a query Q is not a member of the CP, j[R]() is added to g() of the CP.
- *Considering Functional Dependencies:* Having applied the above rules while generating a CP, we may now exclude all columns from the set of grouping attributes which are functionally dependent from other columns of the set of all grouping attributes. Although this step might reduce the number of grouping columns and might affect the time needed for generating a proper query execution time, we do not recommend and further pursue this strategy because the processing of the dependent columns might benefit from an existing ordering with regard to the determining attribute.

To put it into a nutshell, generating a CP results in a huge number of grouping columns in most cases, so that it might be more beneficial to restrict the generation of the CP to a subset of possibly participating tables. Moreover, we must take into consideration that every grouping column which is added to the set of grouping columns may dramatically increase the cardinality of the output data stream, so that generating a CP might increase the cardinality of the processed data and therefore decrease query performance. This might be illustrated by the following example: Consider a table with the two

1. Taking the SQL99 extension of grouping sets() into consideration, we might gain an advantage on the surface referring to the grouping condition of grouping sets((A,B),(B,C)). However, grouping sets are usually resolved into the common aggregation base during evaluation inside the DB engine, so that we do not furthermore consider this alternative specification.

Final edited form was published in "CIKM02: Eleventh ACM International Conference on Information and Knowledge Management. McLean 2002", S. 43–48,
ISBN 978-1-58113-590-9
https://doi.org/10.1145/583890.583897

grouping columns A and B, a fact column C, and the following tuples (1,1,?), (2,2,?), (3,3,?), (4,4,?), (5,5,?). Grouping over A results in 5 tuples of degree 2 (equals processing cost), grouping over B also results in 5 tuples, i.e. costs 5 units based on the raw data as well as on the CP. Thus the total cost of computing a single query based on the CP results in (5/2 queries) + 5 = 7.5 units. Lesson learned: Generating a CP is not always a viable alternative to speed up query performance. Moreover, it might become expensive and must therefore be handled in a cost-based manner, which will be discussed in the following section.

3 GENERATING A SINGLE CP BASED ON THE JOIN LATTICE

For the discussion of generating a single and at the same time most beneficial CP, we refer to a simple star schema consisting of a fact table F(X,Y,...) with primary key columns X and Y and further fact columns which are not considered in our problem. Moreover the schema exhibits two dimension tables D_1(X,A,B,C,D) and D_2(Y,E,F,G,H). For the sake of simplicity, we refer to the following query set consisting of two star queries:

```
SELECT D1.A, D2.E, ...                SELECT D1.D, D2.H, ...
  FROM F, D1, D2                        FROM F, D1, D2
 WHERE F.X = D1.X AND F.Y = D2.Y       WHERE F.X = D1.X AND F.Y = D2.Y
  AND P(D1.B)                           AND P(D1.C)
  AND P(D1.C)                           AND P(D2.F)
  AND P(D2.F)                           AND P(D2.G)
GROUP BY D1.A, D2.E                   GROUP BY D1.D, D2.H
```

Looking at these two incoming queries Q_1 with g(A,E), p(B,C,F), and j(X,Y) and Q_2 with g(D,H), p(C,F,G), and j(X,Y) respectively, the set of our CPs from which we have to pick the optimal one consists of the following combinations:

- *CP_1 with g(A,B,D,E,G,H), p(C,F), and j(X,Y)*
  All tables are used to create the CP so that the list of grouping columns consists of all grouping and local predicate columns according to the rules described in the preceding section. Common local predicates are pushed down into the CP, predicates individual to single queries remain in the compensation.
- *CP_2 with g(A,D,B,Y), p(C), and j(X)*
  CP_3 with g(E,H,G,X), p(F), and j(Y)
  These candidate CPs are considering only the fact table and one of the two dimension tables. Considering the other dimension table is left to the compensation. To be able to delay the join into the compensation, we need the join column in the list of grouping attributes. The common local predicate is again applied in the corresponding CP.
- *CP_4 with g(X,Y), p(), and j()*
  Surprisingly, the fourth candidate CP which has to be considered within the selection process does not contain any join with a dimension table at all and looks very similar to the original fact table. However, this result may be justified from two perspectives. Firstly, the fact table my exhibit further primary key attributes so that this CP candidate would already result in a reduction of cardinality. Secondly, if this candidate CP does correspond to the fact table and it would be elected the best CP, we may derive from this decision that it is not worth generating a CP for the given query scenario and the benefit could be reduced by sharing the table access by the underlying storage manager.

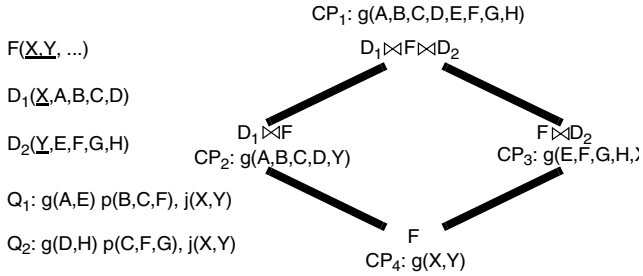


Figure 2: Sample Join Lattice

To summarize, the overall strategy to pick the optimal CP is to take k out of n dimension tables ($0 \le k \le n$) into account and consider all the grouping columns which are referred to in all participating queries with regard to the k dimension tables to form the CP. Regarding a CP, the factor k is called the degree of the CP. Additionally, we have to add the join attributes of the remaining n-k dimension tables to the set of grouping columns. The dependencies between the single candidate CPs may then be modeled using a join lattice which does not only reflect the aggregation combinations (as the classical aggregation lattice does: [10], [4]) but additionally considers the participating dimension tables with their join columns.

Definition: *CP Join Lattice CPJL*

The CPJL is an acyclic graph with 2^n nodes for a given query scenario with a fact table and n dimension tables connected by a 1:N-relationship to the fact table.

A node corresponds to a candidate CP and is characterized by the set of grouping columns constructed for each k, $1 \le k \le 2^n$ according to the following rule:

$$g( ) = \bigcup_{i=1}^{k} (g[D_i]( ) \cup p[D_i]( )) \cup \bigcup_{i=k+1}^{n} j[D_i]( )$$

An edge exists between two nodes if one node considers exactly on more dimension table as the other node.

Figure 2 shows the CPJL regarding the above scenario and the four proposed candidate CPs. It is worth mentioning that although the resulting lattice does exhibit the same shape of the classical aggregation lattice introduced by [10] and referenced in many papers (e.g. [2], [11]), the CPJL is focusing on the consideration of dimension tables, i.e. the width (or degree) of a common predecessor to speed up the following query executions.

Referring to the join lattice for a given schema and query scenario we may now decide on the optimal CP, in picking the one with the lowest total cost to evaluate a set of queries. Section 5 focuses exactly on that problem by giving an algorithm, a cost model, and examples referring to the TPC-H database scenario.

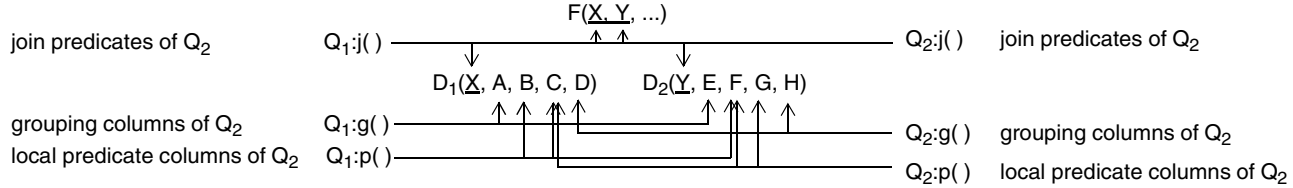


Figure 1: Sample Schema and Query Scenario

Final edited form was published in "CIKM02: Eleventh ACM International Conference on Information and Knowledge Management. McLean 2002", S. 43–48,
ISBN 978-1-58113-590-9
https://doi.org/10.1145/583890.583897

4 GENERATING CPS FOR SNOWFLAKE AND GALAXY SCHEMES

Given a schema with a single fact table and multiple dimension tables, we may construct a CPJL for a given query scenario, estimate the cardinalities, and pick the minimal and therefore most economic CP for further processing. Having a chain of dimension tables like in a snowflake schema or multiple fact tables requires further attention. These problem will be discussed in the context of this section.

Snowflake Schema

The basic difference in building a CP for query scenarios over a snowflake schema consists in an extended rule of aligning join columns. Consider the following situation with a fact table F(X,Y,...), a dimension table D1(X,A,B,C,Y,Z), and two further dimension tables with primary keys as the target of a corresponding foreign key in D_1, i.e. D_{11}(Y,D,E) and D_{12}(Z,F,G). We may in a first step construct a single join lattice for D_{11} and D_{12} with D_1 in the 'role' of the fact table. In a second step this dimensional join lattice is then combined with the regular join lattice where D_1 is the dimension and F the fact table. Figure 3 illustrates this process for the above table scheme.

From this example we may derive the general rule to align join predicates during the construction of the overall lattice:

- *Align Join Columns in a Snowflake Schema:* If a dimension table exhibits a foreign key to another table and the target table is not considered in the construction of the current CP, then the foreign key column(s) of that dimension table must be added to the list of grouping columns.

Galaxy Schema

A data warehouse schema exhibits the pattern of a galaxy scheme if it exhibits more than a single fact table and the dimension tables are shared between the fact tables. Before giving a general strategy to handle schema and query scenarios with multiple fact tables, consider the following scenario with two fact tables F_1(X1,Y1,...), F_2(X2,Y2,...) and a dimension table D(X1,X2,....) which is used by F_1 and F_2 as well. As shown in figure 4, we cannot merge F_1 and F_2 within the context of a CP, because situations may occur where

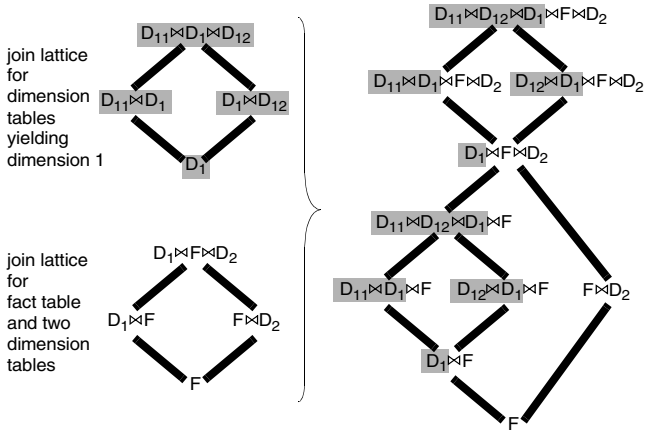


Figure 3: Sample Join Lattice for a Snowflake Schema Scenario

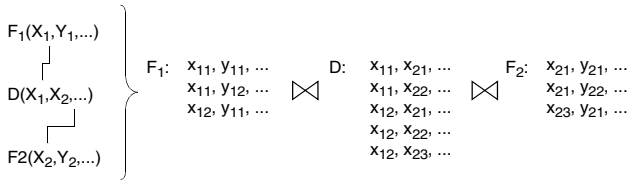


Figure 4: Multiplication of Data within a Galaxy Schema

information gets multiplied or eliminated so that the result will be falsified for compensation queries referring only to a single fact table F_1 or F_2 (Figure 4).

The strategy to generate proper CPs in the context of a galaxy schema relies on the idea of partitioning the schema into regular star or snowflake schemes with a single fact table. These lattices are then independently analyzed with regard to the given query scenario and the optimal CPs are computed. User queries referring to more than one fact table are then rewritten using compensation queries referring to one or more CPs, each coming from a single join lattice. To enable compensation queries using two or more CPs, we have to adapt the rule for aligning join columns appropriately:

- *Aligning Join Columns:* If a dimension table is referred to from more than one fact table either directly or indirectly we have to add the join columns referring to all other fact tables to the list of grouping columns when adding this dimension table to the CP of a fact table.

In the small example above, X_2 of dimension table D has to be added to the set of grouping columns of D when using D to generate a CP in combination with F_1. Similarly, X_1 is a grouping column of a CP starting at F_2.

5 SELECTION ALGORITHM, COST MODEL, AND EVALUATION

The CP join lattice forms the basis for picking the optimal CP for a given query set. In this section, we give a selection algorithm to compute the common predecessor for a given table scenario connected by referential constraints and a set of star queries resulting in the minimum total cost. Moreover, we outline the underlying cost model which is more detailed than the commonly used linear cost model merely considering the number of tuples. Finally, we demonstrate our approach referring to the TPC-H benchmark scenario.

CP Selection Algorithm

Figure 5 gives the algorithm to compute the optimal common predecessor based on a set of fact and dimension tables for a set of queries to by supported be the CP. After generating the corresponding CPJL the cost to compute the CP represented by a node of the CPJL is determined with regard to the underlying cost model. In a second step, the query evaluation costs are estimated under the assumption that the corresponding CP is available. If the total cost is less than the cost estimated for a node in a former run through the CPJL, the current node is temporarily selected as optimal.

Two remarks must be given to the algorithm. First, the function call computing the cost to evaluate a CP represented by node N (ComputeCost(N)) may be equivalently expressed by ComputeCost(N, ∅), i.e. the function computing a query expression (1st parameter) under the assumption of an existing common predecessor (2nd parameter).

Second, the given algorithm returns one single common predecessor. Generating multiple CPs in general is an NP-hard problem ([10]) and must be implemented by applying heuristics or relying on greedy-based algorithms. However, generating a 'reasonable' good solution can be achieved by running the algorithm multiple times

Final edited form was published in "CIKM02: Eleventh ACM International Conference on Information and Knowledge Management. McLean 2002", S. 43–48,
ISBN 978-1-58113-590-9
https://doi.org/10.1145/583890.583897

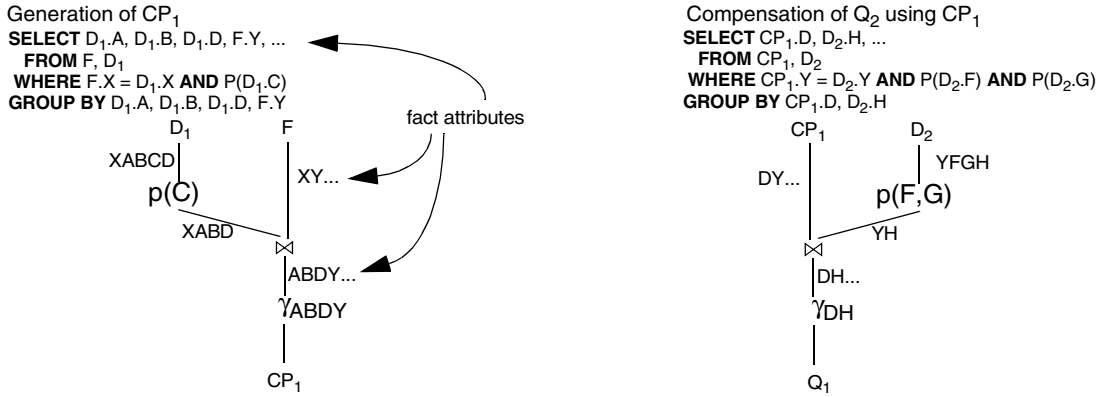


Figure 6: Example to Illustrate the Main Idea of the Cost Model

```
Algorithm: SelectOptimalCommonPrecessor
Input:      F1,..,Fm        // set of fact tables
            D1,...,Dn       // set of dimension tables
            Q1,...,Qp       // query set for computing the CP
Output:     Nopt            // CP with minimal total cost
                            // to evaluate the query set
BEGIN
    Nopt := ∅           // optimal node
    Copt := ∞           // cost of the optimal node

    // Generate the corresponding CPJL following
    // the recipe of section 3 and 4
    CPJL := GenerateJoinLattice({F1,...,Fn}, {D1,...,Dm})

    // Loop through all nodes of the CPJL and pick
    // the optimal table combination forming the CP
    FOREACH N ∈ CPJL
        // Determine the cost to compute the CP
        // represented by node N
        CN := ComputeCost(N)
        CQ := 0

        // Determine the cost to compute all queries
        // based on the currently selected CP
        FOREACH Q ∈ {Q1,...,Qp}
            //Compute cost to evaluate query Q wrt. the
            // currently selected CP denoted by N
            CQ := CQ + ComputeCost(Q, N)
        END IF

        // Remember the optimal node
        IF (CN + CQ < Copt)
            Copt := CN + CQ
            Nopt := N
        END IF
    END FOREACH

    // Return the optimal node of the CPJL denoting the resulting CP
    RETURN (Nopt)
END
```

Figure 5: Algorithm to Generate the Chunking Scheme

and adding each resulting CP to the set of fact tables for the following runs. A single pass of our algorithm shows complexity of *O(n*q)* with *n* as the number of nodes within the generated CPJL and *q* as the number of queries to be considered.

Cost Model

The most crucial point in selecting a common predecessor consists in using an appropriate cost model. Almost every approach in the area of materialized view selection relies on the number of tuples that must be read in order to answer a given query. We extend this model into two directions. First of all, we consider the overall size of the data stream by multiplying the number of tuples with the number of their attributes. Second, we consider all data streams going into a relational operator of the corresponding query graph. This approach is shown for computing CP_1 and query Q_2 from section 3.

Figure 6 illustrates the idea, where the cost of each operator contributes to the overall cost. For applying selection predicates, we assume a table scan, i.e. applying the predicate p(C) during the generation of CP_1 costs 5*card(D_1) units. For the join operator, we may choose between a nested loop and a hash join. The first alternative implies a multiplication of the data stream cardinalities and is omitted in favor of a hash join. For justifying this assumption, we rely on the fact that a machine running a data warehouse does have enough capacity to keep the dimension tables in main memory. A hash join would read the dimension tables during the build phase and the fact table during the probe phase so that we end up in adding the cardinalities of the data streams. For computing CP_1, the cost of the join operator is estimated by 4*card(D_1)*sel(p(C)) + (2+k)*card(F) with k as the number of fact attributes and sel(p(C)) as the selectivity of the predicate over attribute C. For the grouping operator the data stream finally results in (4+k)*card(F)*sel(p(C)) units.

Analogously, the query cost for Q_2 based on CP_1 is estimated based on the size of CP_1. The cardinality of CP_1 however depends on the combination of the distribution over attributes A, B, D, and Y, i.e. card(CP_1) = card(A×B×D×Y). Therefore, we need a mechanism to accurately estimate the cardinality of a data stream after grouping. Unfortunately, this is not easy because usually statistics are gathered on a single attribute level only ([12]). Combining statistics from multiple attributes relies on the independence of the attributes which is not always given. Especially in the warehouse scenario we may easily find clusters, e.g. between brand and color, because a certain brand often has a specific color schema for their products. Many proposals (like [15]) are addressing the problem of estimating the cardinality after a grouping operation over multiple columns and can be incorporated to get reasonable cardinalities of candidate CPs. In the following evaluations, we rely on independence and approximate the cardinality card(A×B×D×Y) by the term MIN(card(A) * card(B) * card(D) * card(Y); size of fact table).

Evaluation using TPC-H Benchmark-Scenario

To demonstrate the selection algorithm, we refer to the TPC-H scenario from section 2 with a mapping of the sample queries of section 3. The table lineitem holds the role of a fact table F, table part is assigned to D_1 and orders corresponds to D_2, each connected by an 1:N-relationship to the fact table. Therefore, the attribute X corresponds to l_partkey/p_partkey and Y is mapped to l_orderkey/o_orderkey. The remaining attributes of D_1 and D_2 are dynamically assigned to attributes of tables part and orders in a way that it yields multiple different scenarios. Figure 7 gives an overview of the roles of the single generic attributes A to H and the cardinalities of the specific attributes taken from TPC-H.

Final edited form was published in "CIKM02: Eleventh ACM International Conference on Information and Knowledge Management. McLean 2002", S. 43–48,
ISBN 978-1-58113-590-9
https://doi.org/10.1145/583890.583897

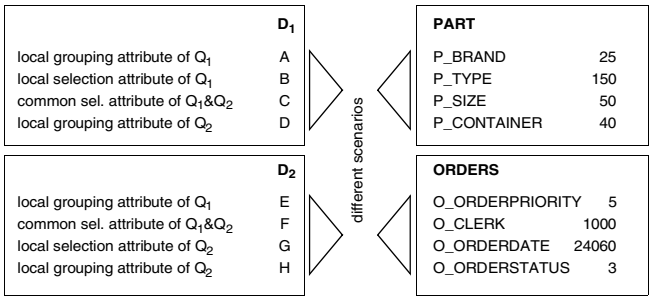


Figure 7: Assigning TPC-H Attributes to Query Skeletons

The first scenario results in a solution which is not encouraging from a pre-computational point of view (figure 8a). The cardinalities of the attributes are assigned so that the optimal solution does not suggest any pre-computation at all. The reason for this behavior is that the common predicates does not exhibit a low cardinality, i.e. a high selectivity (in our case: 1/card()) and – at the same time – attributes contributing to the set of grouping columns for the CP do have a high cardinality thus increasing the cost needed when referring to the materialized CP in the compensation. Changing the situation yields the opposite suggestion (figure 8b). In this case, common predicates have a high selectivity (= low cardinality), grouping attributes have a low cardinality so that the overhead is minimal when using the CP.

From the evaluations, we may learn that every combination is generally possible and the selection of an optimal CP cannot be decided on the schema level, i.e. looking only at the attributes but needs information from the instance level like one-dimensional and multi-dimensional data distributions. These design criteria should be incorporated when materializing summary data within a data warehouse database.

6 SUMMARY AND CONCLUSION

Within this paper we address the problem of generating a common predecessor for multiple queries. In a first step we identified many applications for applying those optimization strategies. Thereafter, the paper discusses several strategies to come up with an optimal predecessor. As we are illustrating within this paper, generating the most general predecessor in a query environment does not always result in the best solution. A proposed algorithm thus takes the overall query into consideration and computes the optimal predecessor for a given set of star queries. Evaluations show that the result heavily depends on the data distribution. Single and multi dimensional statistics are needed to come up with a reliable suggestion. The technique of providing common predecessors which might be beneficially used by multiple queries can be incorporated into many applications like incremental maintenance of materialized views or preparing an application specific extract, i.e. data mart.

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*, pp. 506-521

2 Baralis, E.; Paraboschi, S.; Teniente, E.: Materialized Views Selection in a Multidimensional Database. In: *VLDB'97*, pp. 156-165

3 Chaudhury, S.; Krishnamurthy, R.; Potamianos, S.; Shim, K. Optimizing Queries with Materialized Views. In: *ICDE'95*, pp. 190-200

4 Deshpande, P.M.; Ramasamy, K.; Shukla, A.; Naughton, J.F.: Caching Multidimensional Queries Using Chunks. In: *SIGMOD'98*, pp. 259-270

5 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*, pp. 152-159

6 Gupta, H.; Harinarayan, V.; Rajaraman, A.; Ullman, J.D.: Index Selection for OLAP. In: *ICDE'97*,pp. 208-219

7 Goldberg, J.; Larson, P.-A.: Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. In: *SIGMOD 2001*, pp. 331-342

8 Gray, J.; Reuter, A.: *Transaction Processing Concepts and Techniques*. Morgan Kaufmann, San Mateo (CA), 1993

9 Gupta, A.; Mumick, I. S.: *Materialized Views Techniques, Implementations, and Applications.* Cambridge (Massacusetts), London (England), 1999

10 Harinarayan, V.; Rajaraman, A.; Ullman, J.D.: Implementing Data Cubes Efficiently. In: *SIGMOD'96*, pp. 205-216

11 Lehner, W.; Cochrane, R.; Pirahesh, H.; Zaharioudakis, M.: fAST Refresh using Mass Query Optimization. In: *ICDE 2001*, pp. 391-398

12 Haas, P.J.; Naughton, J.F.; Seshadri, S.; Stokes, L.: Sampling-Based Estimation of the Number of Distinct Values of an Attribute. In: *VLDB'95*, pp. 311-322

13 Sellis, T.: Multiple Query Optimization. In: *ACM TODS 13(1)*, 1988, pp. 23-52

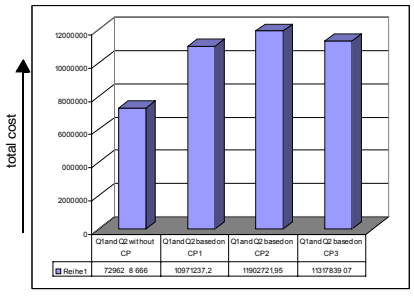
14 Scheuermann, P.; Shim, J.; Vingralek, R.: WATCHMAN: A Data Warehouse Intelligent Cache Manager. In: *VLDB'96*, pp. 51-62

15 Shukla, A.; Deshpande, P.; Naughton, J.F.; Ramasamy, K.: Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies. In: *VLDB'96*, pp. 522-531

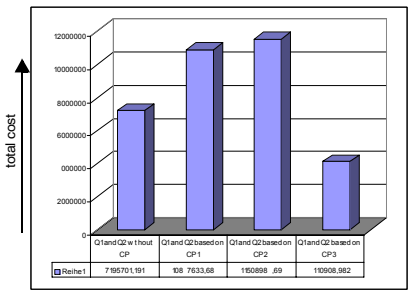
16 Yang, J.; Karlapalem, K.; Li, Q.: Algorithms for Materialized View Design in Data Warehousing Environment. In: *VLDB'97*, pp. 136-145

17 Zaharioudakis, M.; Cochrane, R.; Lapis, G.; Pirahesh, H.; Urata, M.: Answering Complex SQL Queries Using Summary Tables. In: *SIGMOD 2000*, pp. 105-116

18 Zhao, Y.; Deshpande, P.M.; Naughton, J.F.; Shukla, A.: Simultaneous Optimization and Evaluation of Multiple Dimensional Queries. In: *SIGMOD'98*, pp. 271-282



a) do not generate any predecessor



b) precompute the full star join

D_1	PART
A	P_BRAND
B	P_TYPE
C	P_SIZE
D	P_CONTAINER

D_2	ORDERS
E	O_ORDERPRIORITY
F	O_ORDERSTATUS
G	O_ORDERDATE
H	O_CLERK

D_1	PART
A	P_BRAND
B	P_SIZE
C	P_TYPE
D	P_CONTAINER

D_2	ORDERS
E	O_ORDERPRIORITY
F	O_ORDERDATE
G	O_ORDERSTATUS
H	O_CLERK

Figure 8: Evaluation of Different Scenarios