Universität Stuttgart

Fakultät Informatik

Improving the Processing of Decision Support Queries: Strategies for a DSS Optimizer

Authors:
Dipl.-Inform. Holger Schwarz
Dipl.-Inf. Ralf Wagner
Prof. Dr. B. Mitschang

Institute of Parallel and Distributed
High-Performance Systems (IPVR)
Department of Computer Science
University of Stuttgart
Breitwiesenstr. 20-22
70565 Stuttgart
Germany

Improving the Processing
of Decision Support Queries:
Strategies for a DSS Optimizer

H. Schwarz, R. Wagner, B. Mi tschang

Technical Report TR-2001-02
Mai 2001

Improving the Processing of Decision Support Queries: Strategies for a DSS Optimizer

Holger Schwarz, Ralf Wagner, Bernhard Mitschang
Institute of Parallel and Distributed High-Performance Systems, University of Stuttgart
D-70565 Stuttgart, Germany
{hrschwar, wagnerrf, mitsch}@informatik.uni-stuttgart.de

Technical Report TR-2001-02
Department of Computer Science
University of Stuttgart
Mai 2001

Abstract

Many decision support applications are built upon data mining and OLAP tools and allow users to answer information requests based on a data warehouse that is managed by a powerful DBMS. In this paper, we focus on tools that generate sequences of SQL statements in order to produce the requested information. Our thorough analysis revealed that many sequences of queries that are generated by commercial tools are not very efficient. An optimized system architecture is suggested for these applications. The main component is a DSS optimizer that accepts previously generated sequences of queries and remodels them according to a set of optimization strategies, before they are executed by the underlying database system. The advantages of this extended architecture are discussed and a couple of appropriate optimization strategies are identified. Experimental results are given, showing that these strategies are appropriate to optimize query sequences of OLAP applications.

1. Introduction

During the last decade data warehouses turned out to be the common basis for the integration and analysis of data in modern enterprises. Decision support applications are used to analyze data on the operational level as well as on the strategic level. This includes techniques like online analytical processing (OLAP) and data mining. Additional tools are used for the preprocessing and integration of data from different sources.

A lot of work has been done on decision support systems and their optimization. In the field of data mining the focus was on algorithms. There is for example a huge set of algorithms for mining association rules including parallel algorithms [11] [2]. For OLAP applications, several important aspects have been discussed in literature. One is the integration of additional operators into the database system, e.g. the cube operator [9] [10] and its efficient implementation [12]. Query primitives that allow related query plans to share portions of their evaluation are introduced in [33]. Another focus is the use of specific index types in order to optimize typical OLAP queries [18] [19]. There has also been a lot of work in the area of using materialized views to answer decision support queries and the maintenance of these materialized views [3] [26] [27] [32]. Our work is recognizing this previous work, but also complementing it. We are not aware of any comparable approach that involves a separate optimizer in between the decision support tool and the data warehouse database system (DWDBS) as we suggest it.

In this paper, we focus on a commonly used processing model for decision support systems (DSS). According to this model, the application generates a sequence of SQL statements, which is processed by the DWDBS. The result tables of the statements in a sequence build the basis for the final result that is presented to the user as reply to his/her information request. SQL is used as query language because most data warehouses are based on a relational or extended relational database system. As the information requests of the users are likely to be very complex, many applications produce sequences of rather simple statements for each request in order to reduce the complexity of the query generation process and in order to preserve portability to other database systems.

We analyzed tools that work according to this model and found that there are many possibilities to improve the query sequences they generate and the way they are executed. We will show that several of these approaches are independent of the application and its special query generation process. Hence, they could be used for many decision support applications that are run on a data warehouse. This leads us to an optimized system architecture. The central element of this architecture is a **DSS optimizer** that accepts sequences of SQL statements. The sequences are processed according to a set of optimization strategies. An optimized version of the sequence is sent to the DWDBS and executed there. As in the original model, the application uses the partial results in order to produce the final result for an information request. The strategies we suggest for the DSS optimizer include the combination of a sequence of statements into a single complex statement, semantic rewrite strategies, parallel execution of statements in a query sequence as well as providing additional statistical information for the query optimizer of the DWDBS.

This report is organized as follows: In Section 2 we describe our application scenario and the common architecture of decision support systems. The optimized architecture that includes a DSS optimizer as an additional component is introduced in Section 3. In Section 4 we discuss some strategies that are appropriate for this DSS optimizer. The results of our performance analysis assessing the benefits of these strategies are explained in Section 5. Finally, our conclusions are given in Section 6.

2. Application Scenario

In this section we describe some basic aspects of data warehousing, decision support systems and the OLAP application scenario we have chosen for our investigations. We present some examples for typical information requests and the common architecture of systems that process such requests.

2.1. Data Warehousing

Data warehousing has gained more and more importance for decision support in organizations and enterprises [15] [25]. Most medium-sized and large organizations integrate data that is relevant for decisions and that originate from different data sources into a data warehouse. Due to the huge amount of data, usually parallel, (object-) relational database systems are deployed. In addition, several tools are needed for the extraction and cleansing process as well as for the analysis of the data.

The analysis of data has two main aspects. On the one hand, OLAP tools are used to view the data along different dimensions [4]. Typical dimensions that are relevant for an enterprise are time, customers, suppliers and products. The users should know which information they are about to retrieve from the data warehouse. In knowledge discovery, on the other hand, data mining tools are able to detect correlations and to find previously unknown pattern in data. It is possible, for example, to determine correlations between the consuming behavior of a certain group of customers and some characteristics of these customers like age or profession. Examples for OLAP tools are MicroStrategy [16] and BusinessObjects [1] whereas Darwin [28], Enterprise Miner [24] and Intelligent Miner [29] are representatives for data mining tools.

Typical data models for data warehouses are called star schema and snowflake schema. For our experiments, we have chosen the snowflake schema given in Figure 1 that shows some important aspects of a retailer. Our schema is based on the TPC-H Benchmark [30], which comprises decision support SQL queries. The original schema consists of eight tables describing a typical scenario of a retailer. The retailer receives orders (table ORDERS) that consist of single items (LINEITEM). The orders are sent by customers (CUSTOMER), which reside in a nation (NATION) and region (REGION). The items refer to parts (PART) delivered by suppliers (SUPPLIER), which reside in a nation and region, too. Finally, the parts delivered by a supplier are given by the joint table PARTSUPP.

We used a modified schema for our experiments. There are two main reasons for that:

- The OLAP tool we used for our experiments needs a special type of schema, a snowflake schema with partially redundant dimension tables. The necessary modifications are just formal, and easily achievable by syntactical equivalence transformations. There are no semantic changes.
- The time dimensions are extended in two ways. First, the date field of the original schema is split into separate tables for days, weeks, months and years. Second, additional information is provided for each unit of time, e.g. the last day, week or month for a given date. These extensions make days, weeks, months etc. explicitly available for queries and enable for example the comparison of a given month with the respective month in the preceding years.

The modified schema has more fact and dimension tables than the original schema. Figure 1 depicts the part of the modified schema, which is relevant for our example queries. The main fact table is shown in the middle of the picture. Some of the dimension tables and connections to further tables are given around this fact table.

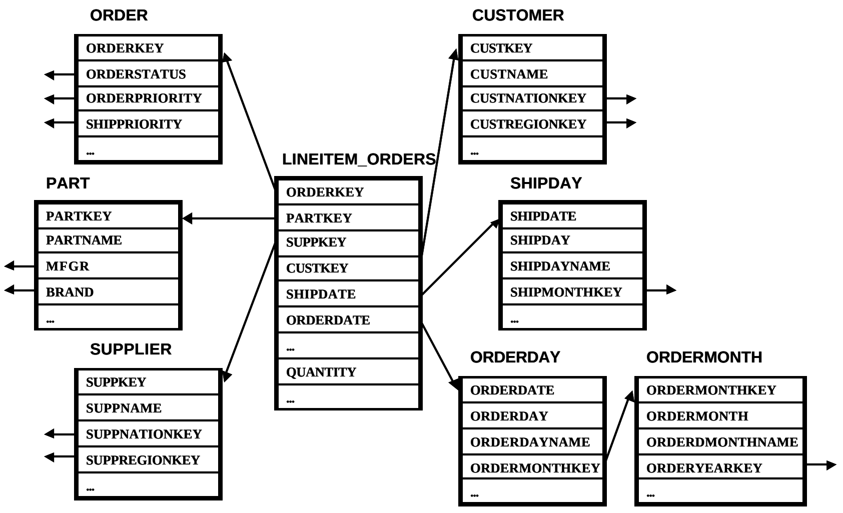


Figure 1. Snowflake Schema based on TPC-H

2.2. OLAP Scenario

We have chosen a typical OLAP scenario of a retail company for our experiments. This includes some information requests a user could define and process by means of an OLAP tool. The selection of requests is based on the following requirements:

- The requests have to be typical questions a retailer would ask based on the information available in the data warehouse.
- It must be feasible to model the requests with currently available OLAP tools and it must be feasible to answer these questions based on our schema.
- The complexity of selected requests should be similar to the complexity of business questions in the TPC-H benchmark.

Three typical information requests that meet these requirements are shown in Figure 2. They are important for the management of a retailer as the questions belong to the categories merchandise management or customer relationship management [14] [23]. The schema we presented above includes the relevant information that is necessary to answer these requests.

Information request A:
Which are the top products whose number of sold pieces in the months chosen by the user compared to the respective month ago has increased mostly?

Information request B:
What is the distribution of the number of sold products over the different countries for products and years chosen by the user?

Information request C:
Which are the top customers who bought products in the last three years of a minimum total amount chosen by the user and who show the lowest standard deviation of the totals in these years?

Figure 2. Selected Information Requests

2.3. Architectural Issues

Numerous decision support applications are based on the system architecture given in Figure 3. The end users specify the information they need by means of a graphical user interface. For example, they have to define the relevant data and the necessary calculations on these facts as well as criteria for filtering and the way results should be presented. These requirements are used by the application to generate a sequence of SQL queries. Meta data is used by the query generator in order to produce the appropriate query sequences. In an OLAP application the meta data include information about available fact tables and the hierarchical dependencies of attributes in one dimension. As soon as the query generator produced a sequence of SQL queries, these queries are sent to the data warehouse database system one after the other. The application reads and processes partial results from the DWDBS and finally presents the result to the end user.

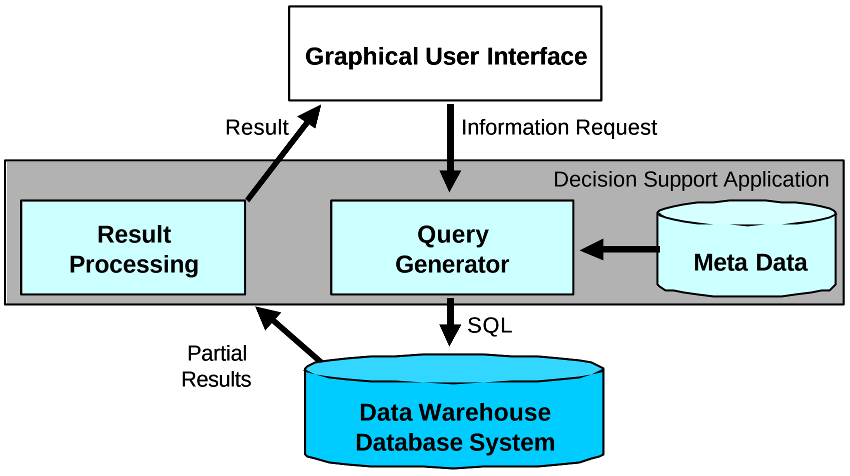


Figure 3. Standard System Architecture

Many typical decision support applications generate sequences of SQL queries. For information request A in our scenario such a sequence is given in Figure 4. This query sequence (QS) was produced by an OLAP tool. The Figure only shows the four insert statements. The create table statements and other details that are also part of the generated sequence are omitted here for the sake of readability. The first statement (Q1) produces the temporary table *A1* that includes the sum of the fact *quantity* for all parts in January and February 1994. The second statement (Q2) only differs in that it sums the same quantity for the preceding months of January and February 1994. The purpose of the third query (Q3) is mainly to calculate the absolute and relative increase of the quantity for each part based on the values of Q1 and Q2. Finally, Q4 selects all parts from Q3 according to the given ranking criterion and provides the result data in table *A4*.

This sample query sequence consists of four statements. In general, the conversion of OLAP information requests into sequences of SQL statements results in sequences of different complexity, which ranges from just one to eight queries for the information requests we have chosen here. According to our experience, it is very likely that relevant business questions result in query sequences with several statements, sometimes even more than 20. Hence, the three information requests given in Figure 2 represent important types of OLAP queries and are relevant in order to judge optimization strategies.

```
INSERT INTO A1 (orderyearkey, ordermonthkey, partkey, sumquantity)
  SELECT od.orderyearkey, od.ordermonthkey, lo.partkey, SUM(lo.quantity)
  FROM   lineitem_orders lo, orderday od
  WHERE  od.orderdate = lo.orderdate  AND od.ordermonthkey IN (199401,199402)
  GROUP BY od.orderyearkey, od.ordermonthkey, lo.partkey;                        Q1
```

```
INSERT INTO A2 (ordermonthkey, partkey, sumquantity)
  SELECT od.ordermonthkey, lo.partkey, SUM(lo.quantity)
  FROM   lineitem_orders lo, orderday od
  WHERE  od.lastmonthdate = lo.orderdate AND od.ordermonthkey IN (199401, 199402)
  GROUP BY od.ordermonthkey, lo.partkey;                                          Q2
```

```
INSERT INTO A3 (ordermonthkey, ordermonthname, orderyearkey, orderyear, partkey, partname,
                 sumquantity, lmsumquantity, incrquantity, incrquantity2)
  SELECT om.ordermonthkey, om.ordermonthname, oy.orderyearkey, oy.orderyear, pa.partkey,
         pa.partname, A1.sumquantity, A2.sumquantity, A1.sumquantity - A2.sumquantity,
         (A1.sumquantity - A2.sumquantity) / A2.sumquantity
  FROM  A1,A2, ordermonth om, orderyear oy, part pa
  WHERE A1.ordermonthkey = A2.ordermonthkey AND A1.partkey = A2.partkey
  AND   A1.ordermonthkey = om.ordermonthkey                                       Q3
  AND   A1.orderyearkey = oy.orderyearkey AND A1.partkey = pa.partkey;
```

```
INSERT INTO A4 (ordermonthkey, ordermonthname, orderyearkey, orderyear, partkey, partname,
                 sumquantity, lmsumquantity, incrquantity, incrquantity2)
  SELECT A3.ordermonthkey, A3.ordermonthname, A3.orderyearkey,
         A3.orderyear, A3.partkey, A3.partname, A3.sumquantity,
         A3.lmsumquantity, A3.incrquantity, A3.incrquantity2
  FROM   A3                                                                       Q4
  WHERE  A3.incrquantity2 >= 98;
```

Figure 4. Query Sequence for Request A

3. Optimized System Architecture

In the standard architecture presented in Figure 3 the performance of a decision support application mainly depends on the capabilities of the query generator and on the query optimization and execution capabilities of the DWDBS. Therefore, if one envisages performance problems there are two different ways to go: On the one hand, one could improve the query generator and on the other, one could improve the query optimizer of the DWDBS.

Improving the query generator means to enhance the set of rules according to which the queries are generated. This task depends on the underlying DWDBS. The optimum query sequence for IBM DB2 could look quite different compared to the optimum sequence for Oracle or SQL Server. Since almost all decision support applications aim to support several database systems, the optimization strategies of the query generator have to be developed for each dedicated database system. Furthermore, this has to be done for each decision support application separately. Hence, there could be different optimization strategies for every combination of a decision support application and the underlying DWDBS.

Improving the query optimizer of the DWDBS is also a very difficult task. New query optimization strategies for decision support that are implemented in a database system should on the one hand be based on the analysis of a couple of decision support applications. On the other hand, the developers have to show that the new optimizations do not negatively influence the performance of other types of applications or even deteriorate the performance of the optimizer itself.

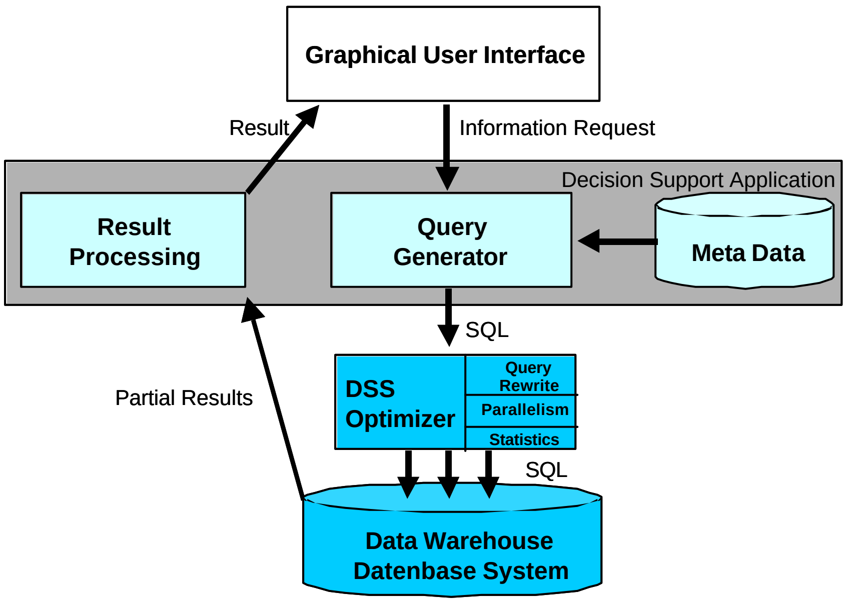


Figure 5. Optimized System Architecture

In summary, both starting-points for performance enhancements of decision support applications in a standard architecture turn out to be very difficult. The complexity results from the heterogeneity of applications and database systems as well as from the complexity of query optimizers that are used in commercial database systems.

We suggest an optimized system architecture for decision support which is shown in Figure 5. The main idea of this architecture is a DSS optimizer as an additional system component between the query generator and the DWDBS. This optimizer accepts query sequences from different applications. It transforms these sequences into optimized versions of query sequences, which are then sent to the DWDBS. The DSS optimizer is neither part of a special application nor of the database system. The main advantages of the extended architecture are as follows:

- In a typical data warehouse environment, several decision support applications access data in the warehouse. The DSS optimizer offers performance enhancements for all of these applications.
- The DSS optimizer is not part of a decision support application. Therefore, its development and enhancement is not part of the application development. It is done only once and not especially for each application.
- The DSS optimizer offers performance enhancements for existing decision support applications without the need to change the query generation process of these applications. They benefit from the optimizer simply by directing their query streams to it instead of sending them directly to the DWDBS.
- Using the DSS optimizer reduces the need for each application to consider special capabilities of the underlying database system. The query generators of the applications can mostly be kept independent of the database system, thus they are less complex.
- The DSS optimizer may include optimization strategies that are offered by some database systems as well as strategies that are not supported by state-of-the-art database systems. Multi-Query-Optimization is an example for an area where current systems offer only little support.

In summary, the optimized system architecture offers support for many decision support applications and many database systems without additional development effort for each of the applications or database systems. In the following chapter, we further discuss a couple of essential optimization strategies for the DSS optimizer. All strategies are demonstrated in our OLAP scenario.

4. Strategies for the DSS Optimizer

There are a couple of strategies that offer performance enhancements for query sequences which are generated by decision support tools. For example, one could rewrite the query sequence. This ranges from changes to some queries of the sequence up to the combination of all queries of the sequence into one single query. Other alternatives are to run several queries in the sequence in parallel or to feed the query optimizer of the DWDBS with supplementary statistical information, which is required to find appropriate query plans for all queries in a sequence. Additionally, changes to the schema of the data warehouse, like e.g. additional indexes, aggregate or partition tables, could support a given query sequence. In this section, we discuss these optimization strategies. For each of them we argue whether it is an appropriate strategy for the DSS optimizer or not. The goal is to incorporate all optimizations into the DSS optimizer that need no information about the application and little information about the underlying database system. For all strategies we identified as appropriate, we discuss the main aspects of their implementation.

Not all optimization strategies mentioned here are new. The contribution of this paper is not to find new algorithms for a database optimizer, but to combine existing technology in an additional system component that offers performance advantages for a huge range of applications and that is independent of the concrete application and the DWDBS. Currently, none of the existing systems follows this approach.

4.1. Rewrite Strategies for Query Sequences

In this section we describe three different strategies to rewrite a given query sequence.

Strategy I: Single-Query (SQ)

Typical query sequences that are produced by decision support applications can be rewritten as a single and in most cases more complex query. A straightforward method to achieve this single-query is as follows: Starting with the second query in the sequence, replace the occurrence of each temporary table in the FROM clause by the complete SELECT statement that generates data for this temporary table. If we use this simple syntactic replacement algorithm for the query sequence in Figure 4, we first have to look at Q2. As this one does not use any temporary table, we have to examine query Q3. It needs data from table *A1* as well as from *A2*. Therefore, the FROM clause is changed and *A1* is replaced by the SELECT statement of Q1. The same replacement is necessary for *A2*. The new query for *A3* does not include any references to temporary tables. Hence, we can proceed with the last query in this sequence. It is only based on the content of *A3*. Its purpose is to filter out the relevant data according to the ranking criterion. We replace *A3* in the FROM clause of query Q4 by the new version of the query for *A3*. The resulting single-query for information request A is shown in Figure 6. The corresponding single-query for information request C is given in the appendix.

This type of query rewrite that is similar to view expansion is an appropriate strategy for the DSS optimizer because it does not require any knowledge about the application that generated the query sequence or about the DWDBS. The rewrite strategy is only based on the list of queries that constitute the given query sequence.

Why should this single-query run faster than the original sequence of queries? There are two main arguments. First, there is less overhead for creating temporary tables and for inserting data into these tables. Only one temporary table is explicitly generated and filled with data, whereas all other temporary results are handled by the database system as part of the query execution. In our experiments, some temporary tables contained several millions of rows. These tables were filled with about 8000 rows per second. Thus, loading temporary tables represents significant overhead for some of the query sequences. There is some additional overhead for the creation of these temporary tables to be taken into account.

Second, there are more chances for query optimization by the DWDBS. In case of a query sequence, the statements of the sequence are sent to the DWDBS one after the other. Hence, the query plan for each statement is generated without any knowledge about the other statements in the sequence. If the complete sequence is combined into one single statement the query optimizer has the whole picture. Therefore, the optimizer might have a chance to exploit common sub-expressions among the query portions that previously belonged to separate queries.

```
INSERT INTO A4 (ordermonthkey, ordermonthname, orderyearkey, orderyear,
                partkey, partname, sumquantity, lmsumquantity,
                incrquantity, incrquantity2)

  SELECT A3.ordermonthkey, A3.ordermonthname, A3.orderyearkey,
         A3.orderyear, A3.partkey, A3.partname, A3.sumquantity,
         A3.lmsumquantity, A3.incrquantity, A3.incrquantity2
  FROM
    (SELECT om.ordermonthkey, om.ordermonthname, oy.orderyearkey,
            oy.orderyear, pa.partkey, pa.partname, A1.sumquantity,
            A2.sumquantity, A1.sumquantity - A2.sumquantity
            AS incrquantity,
            (A1.sumquantity - A2.sumquantity) /                    from Q3
            A2.sumquantity AS incrquantity2
     FROM
      (SELECT od.orderyearkey, od.ordermonthkey,
              lo.partkey, SUM(lo.quantity)
       FROM   lineitem_orders lo, orderday od
       WHERE  od.orderdate = lo.orderdate                          from Q1
       AND    od.ordermonthkey IN (199401, 199402)
       GROUP BY od.orderyearkey, od.ordermonthkey, lo.partkey
      ) AS A1 (orderyearkey, ordermonthkey, partkey, sumquantity),
      (SELECT od.ordermonthkey, lo.partkey, SUM(lo.quantity)
       FROM   lineitem_orders lo, orderday od
       WHERE  od.lastmonthdate = lo.orderdate                      from Q2
       AND    od.ordermonthkey IN (199401, 199402)
       GROUP BY od.ordermonthkey, lo.partkey
      ) AS A2 (ordermonthkey, partkey, sumquantity),
        ordermonth om, orderyear oy, part pa
     WHERE A1.ordermonthkey = A2.ordermonthkey
     AND A1.partkey = A2.partkey AND A1.ordermonthkey = om.ordermonthkey
     AND A1.orderyearkey = oy.orderyearkey AND A1.partkey = pa.partkey
     ) AS A3 ( ordermonthkey, ordermonthname, orderyearkey, orderyear,
               partkey, partname, sumquantity, lmsumquantity,
               incrquantity, incrquantity2)
                                                                   from Q3
  WHERE A3.incrquantity2 >= 98;
```

Figure 6. Single-Query for Request A

Given these advantages of a single-query, it seems to be a good idea to combine queries of a sequence into a single one in general. Nevertheless, the task is not that easy for the DSS optimizer. We have further analyzed the queries in our application scenario based on the query plans the optimizer of DB2 generated for our experiments. Table 1 shows how they differ in the number of joins and sort operations. The values given here especially show that the straightforward combination of the original query sequence into one SQL query can result in a large number of joins and sorts. For information request C, the single-query (SQ) needs 33 joins and 37 sort operations. The corresponding query plan has more than 200 nodes. This complexity is not easy to handle for the optimizer of the DWDBS. Its output might be a rather 'bad' query plan. As one example, the number of join orders increases exponentially with the number of joins. With more than 10 joins, it is likely that the optimizer is not able to consider all relevant orders. Additionally, it might not detect all common sub-expressions that are part of the query and that could be combined. The appropriate and manageable complexity of queries depends on the characteristics of the optimizer used in the DWDBS.

So far, our discussion has shown that in some cases a sequence of queries might run faster than the combined single-query and that in other situations this might not be true. Our experiments show examples for both cases. Hence, the DSS optimizer has to decide for each query sequence whether it should be combined or not.

Table 1. Characteristics of different Queries

Optimization Strategy	Information Request A		Information Request C	
	JOINS	SORTS	JOINS	SORTS
QS	max. 5	max. 10	max. 6	max. 8
QS + MergeSelect	-	-	max. 3	max. 3
QS + WhereToGroup	-	-	max. 4	max. 3
SQ	7	14	33	37
SQ + MergeSelect	-	-	13	14
SQ + WhereToGroup	-	-	16	16

Strategy II: Partial Combination

One alternative is not to combine the whole sequence into a single-query but to merge it into a couple of queries. As we have argued, one single-query could be too complex for the optimizer of a DWDBS to find a good execution plan. If the DSS optimizer generates a small number of semi-complex queries, this new sequence could be more efficient.

For some query sequences it is not possible to generate an equivalent single-query, because there are dependencies within the sequence. This appears in the case of ranking. In the query sequence shown in Figure 4 the last query Q4 selects data according to a filter criterion. For the sake of simplicity, we assumed that the filter value is known in advance. Depending on the way the user defined the information request and the query generation process of the application, this filter predicate might be determined based on data in *A3*. In this case, the last query of the sequence could include a variable instead of the filter predicate. Hence, the partial combination strategy is applicable. The DSS optimizer could combine Q1, Q2 and Q3 as described for Strategy I and leave Q4 as is. The same holds for the last query of QS for information request C.

This strategy is suitable for the DSS optimizer because it is only based on the list of queries that constitute the given query sequence. The optimization task is to find the proper points where to cut off the original sequence. This could be done based on dependency graphs as given in Figure 9. A directed dependency graph G consists of nodes Q_i, which represent the queries. An edge from Q_k to Q_l denotes that the result of Q_k is used by query Q_l. The queries of a subset G' of G can be combined if the following conditions hold:

- G' has only one exit point Q_j, i.e. the result of only one of the queries in G' is accessed by queries in G that are not part of G'.
- Q_j depends on all other queries in G' by direct or transitive dependencies, i.e. G' is a coherent graph.
- There are no cyclic dependencies in G', in particular no query in G' depends on Q_j.

For information request C the partial combination strategy leads to the following set of alternative query sequences: Q1, {Q2, …, Q7}, Q8; Q1, Q2, {Q3, …, Q7}, Q8; Q1, …, Q3, {Q4, …, Q7}, Q8; Q1, …, Q4, {Q5, …, Q7}, Q8; Q1, …, Q5, {Q6, Q7}, Q8 and Q1, …, Q4, Q6, {Q5, Q7}, Q8. In this list {Q3, ..., Q7} for example denotes the combination of all queries for the temporary tables C3 through C7 into one query. The query sequences we present in this paper benefit more from other types of query rewrite as will be given in the next section.

Strategy III: Semantic Rewrite

Another deciding factor for the DSS optimizer is based on supplementary analysis of the original query sequence. One method to minimize the runtime of query execution is to minimize the number of temporary tables in the whole sequence, thus removing the overhead for writing to and reading from temporary tables. It is also important to avoid repeated references to temporary tables when composing the single-query. Most temporary tables imply joins between underlying tables. Repeated referencing of temporary tables in different queries of a sequence implies a redundant computation of these temporary tables via nested SQL statements when composing the single-query.

There is a large set of transformations that could be applied to query sequences. We only give some examples of rather simple ones here and describe them in an informal way. As it is not our intention to give an extensive enumeration of transformations, we focus on some transformations that we were able to apply to the query sequences for our sample information requests. A detailed analysis and formal description is subject of our current work. For all transformations, we suppose that only the result of the last query in a sequence is relevant for the application.

- There are Queries, which restrict one single previous temporary table by additional projections and/or selections. These projections and/or selections can be moved into the foregoing query. In the first case, the projections of both queries are concatenated (*ConcatProject*), whereas this is done for both selections in the second place (*ConcatSelect*).
- Queries with identical FROM, WHERE, GROUP BY, HAVING and ORDER BY clauses can be merged into one query by combining the SELECT clauses. We refer to this transformation as *MergeSelect*.
- Queries with identical SELECT, FROM, GROUP BY, HAVING and ORDER BY clauses can sometimes be merged into one query by concatenating the WHERE predicates (*MergeWhere*). If the queries differ only in a selection on the same column, they can be combined by replacing parts of the WHERE clause by an additional grouping on this column (*WhereToGroup*).
- Queries with identical SELECT, FROM, WHERE, GROUP BY and ORDER BY clauses can be merged into one query by combining the HAVING predicates using logical operations (*Merge-Having*).
- In some queries joins with temporary tables act like filters. These joins can be moved to another query in the sequence (*MoveJoin*).

Based on these rules, we generated two modified query sequences for information request C. *QS+MergeSelect* is shown in Figure 7. We briefly describe the ratio behind the transformations that are marked with arrows.

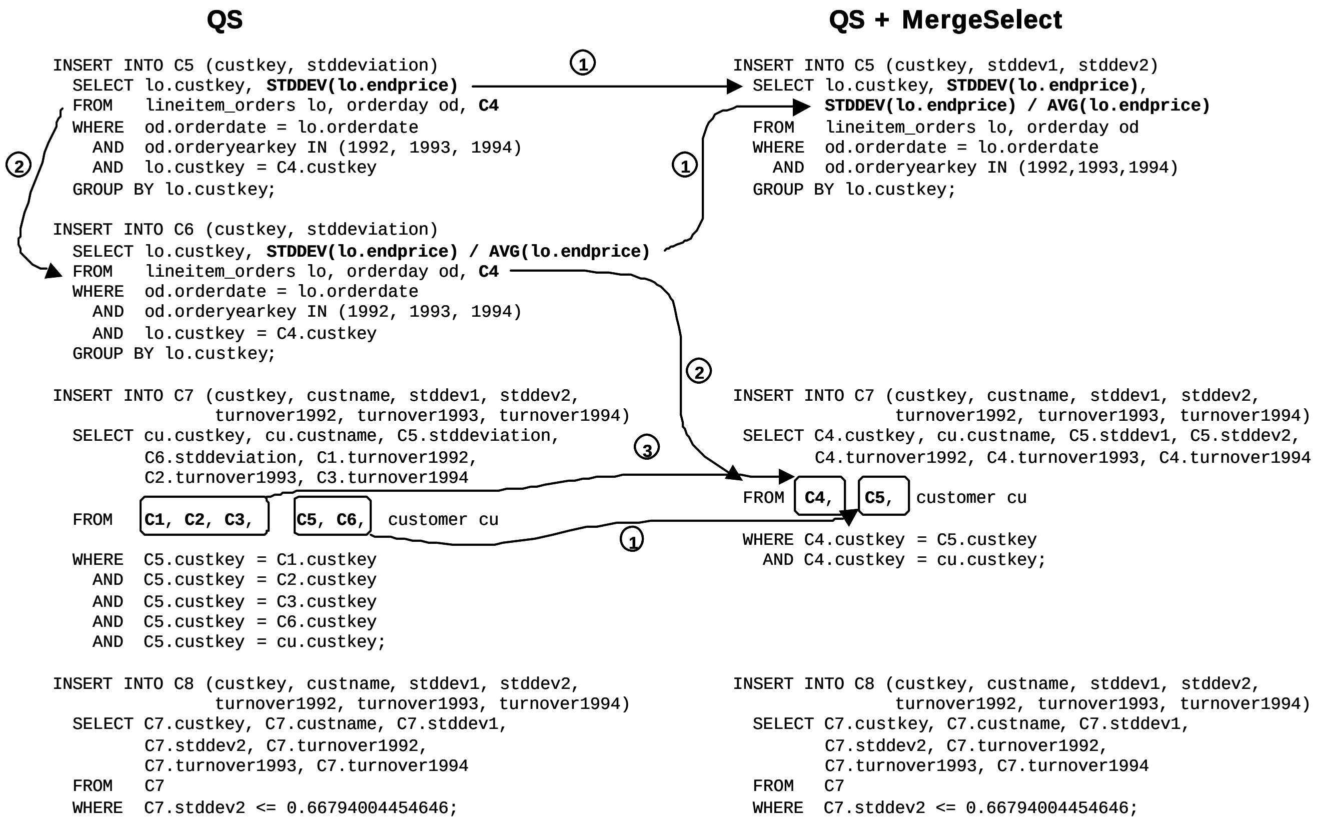
(1) *MergeSelect*
The queries for *C5* and *C6* only differ in the SELECT clause. Using the *MergeSelect* transformation results in a new query for *C5*. It includes the calculation of both necessary standard deviations. In the new query for *C7* only the new table *C5* is used instead of *C5* and *C6*. The join of *lineitem_orders* and *orderday* is now performed only once.

(2) *MoveJoin*
The only purpose of the join with *C4* in the queries for *C5* and *C6* is to filter out customers that are not relevant for further analysis. No column of *C5* or *C6* is based on data in C4. In the modified sequence the joins with *C4* are moved to the query that delivers *C7*. As a result, the number of references to *C4* is cut down.

(3) *MoveJoin*
The temporary table *C4* includes all results of table *C1*, *C2* and *C3* but with additional filtering on the turnover per customer. In *QS* this filter is applied in *C5* and *C6* by joins with *C4*. In the modified sequence, this filter is applied in the query for *C7* as described for transformation (1). Therefore, the join with *C4* replaces the access to *C1*, *C2* and *C3* in the new query for *C7*.

The main parts of the second transformed sequence for information request C are given in Figure 8. It includes the same modifications as *QS+MergeSelect*. The additional transformation is marked as (4). The ratio behind this modification is as follows: The queries for *C1*, *C2* and *C3* of *QS* only differ in the WHERE clause. They can be combined into the new query for *C1* with an additional grouping on *orderyearkey*. Groups with a turnover less than 500000 are filtered. The second query of *QS+WhereToGroup* selects all customers for which a turnover exists for 1992, 1993 and 1994. With this transformation, the number of references to *lineitem_orders* and *orderday* as well as the number of joins between these two tables is reduced.

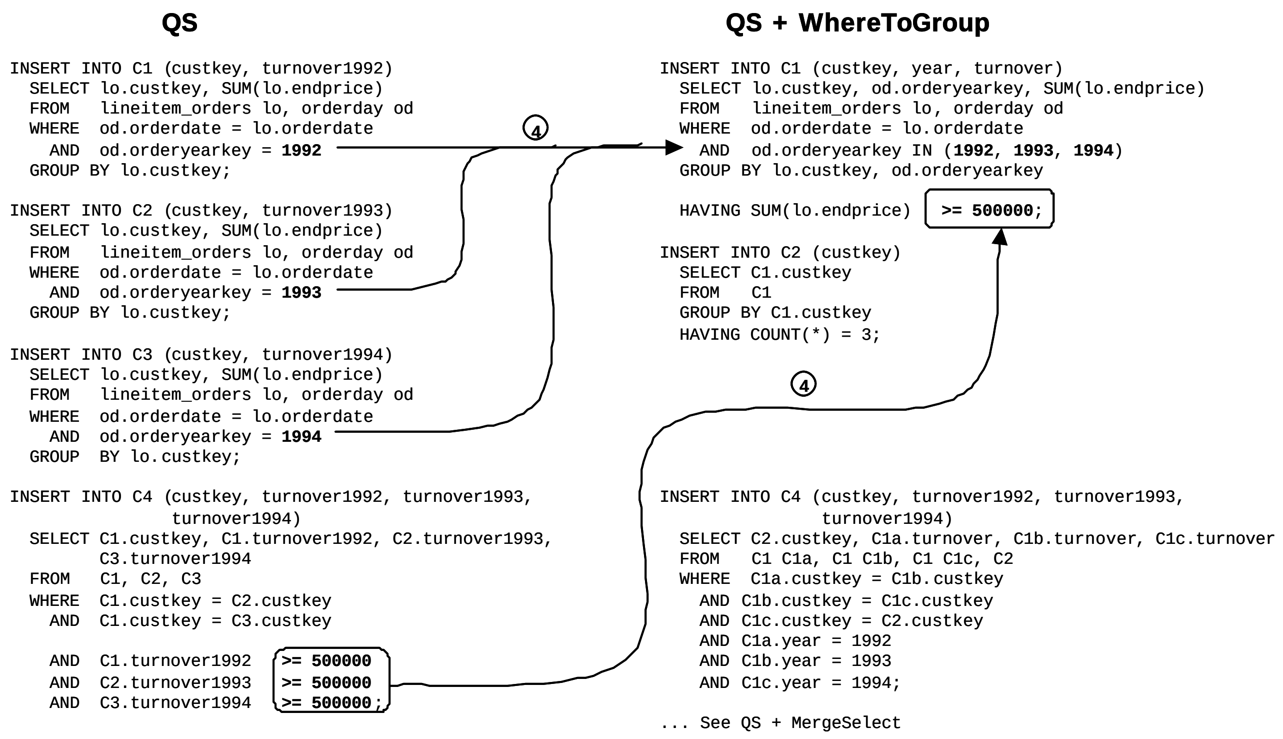
Obviously, these rewrite strategies can also be combined with the single-query strategy and the partial combination strategy. We refer to the queries resulting from these hybrid strategies as *SQ+MergeSelect* and *SQ+WhereToGroup*, where each respective query sequence is combined into one query. As a consequence, the number of join and sort operations in the rewritten queries drop drastically (more than 50%) as can be seen in Table 1.



```
INSERT INTO C5 (custkey, stddeviation)
  SELECT lo.custkey, STDDEV(lo.endprice)
  FROM   lineitem_orders lo, orderday od, C4
  WHERE  od.orderdate = lo.orderdate
    AND  od.orderyearkey IN (1992, 1993, 1994)
    AND  lo.custkey = C4.custkey
  GROUP BY lo.custkey;

INSERT INTO C6 (custkey, stddeviation)
  SELECT lo.custkey, STDDEV(lo.endprice) / AVG(lo.endprice)
  FROM   lineitem_orders lo, orderday od, C4
  WHERE  od.orderdate = lo.orderdate
    AND  od.orderyearkey IN (1992, 1993, 1994)
    AND  lo.custkey = C4.custkey
  GROUP BY lo.custkey;

INSERT INTO C7 (custkey, custname, stddev1, stddev2,
                turnover1992, turnover1993, turnover1994)
  SELECT cu.custkey, cu.custname, C5.stddeviation,
         C6.stddeviation, C1.turnover1992,
         C2.turnover1993, C3.turnover1994
  FROM   C1, C2, C3,  C5, C6,  customer cu
  WHERE  C5.custkey = C1.custkey
    AND  C5.custkey = C2.custkey
    AND  C5.custkey = C3.custkey
    AND  C5.custkey = C6.custkey
    AND  C5.custkey = cu.custkey;

INSERT INTO C8 (custkey, custname, stddev1, stddev2,
                turnover1992, turnover1993, turnover1994)
  SELECT C7.custkey, C7.custname, C7.stddev1,
         C7.stddev2, C7.turnover1992,
         C7.turnover1993, C7.turnover1994
  FROM   C7
  WHERE  C7.stddev2 <= 0.66794004454646;
```

```
INSERT INTO C5 (custkey, stddev1, stddev2)
  SELECT lo.custkey, STDDEV(lo.endprice),
         STDDEV(lo.endprice) / AVG(lo.endprice)
  FROM   lineitem_orders lo, orderday od
  WHERE  od.orderdate = lo.orderdate
    AND  od.orderyearkey IN (1992,1993,1994)
  GROUP BY lo.custkey;

INSERT INTO C7 (custkey, custname, stddev1, stddev2,
                turnover1992, turnover1993, turnover1994)
 SELECT C4.custkey, cu.custname, C5.stddev1, C5.stddev2,
        C4.turnover1992, C4.turnover1993, C4.turnover1994
 FROM C4,  C5,  customer cu
 WHERE C4.custkey = C5.custkey
   AND C4.custkey = cu.custkey;

INSERT INTO C8 (custkey, custname, stddev1, stddev2,
                turnover1992, turnover1993, turnover1994)
  SELECT C7.custkey, C7.custname, C7.stddev1,
         C7.stddev2, C7.turnover1992,
         C7.turnover1993, C7.turnover1994
  FROM   C7
  WHERE  C7.stddev2 <= 0.66794004454646;
```

Figure 7. Modified Sequences for Request C (MergeSelect)



```
INSERT INTO C1 (custkey, turnover1992)
  SELECT lo.custkey, SUM(lo.endprice)
  FROM   lineitem_orders lo, orderday od
  WHERE  od.orderdate = lo.orderdate
    AND  od.orderyearkey = 1992
  GROUP BY lo.custkey;

INSERT INTO C2 (custkey, turnover1993)
  SELECT lo.custkey, SUM(lo.endprice)
  FROM   lineitem_orders lo, orderday od
  WHERE  od.orderdate = lo.orderdate
    AND  od.orderyearkey = 1993
  GROUP BY lo.custkey;

INSERT INTO C3 (custkey, turnover1994)
  SELECT lo.custkey, SUM(lo.endprice)
  FROM   lineitem_orders lo, orderday od
  WHERE  od.orderdate = lo.orderdate
    AND  od.orderyearkey = 1994
  GROUP  BY lo.custkey;

INSERT INTO C4 (custkey, turnover1992, turnover1993,
                turnover1994)
  SELECT C1.custkey, C1.turnover1992, C2.turnover1993,
         C3.turnover1994
  FROM   C1, C2, C3
  WHERE  C1.custkey = C2.custkey
    AND  C1.custkey = C3.custkey

    AND  C1.turnover1992  >= 500000
    AND  C2.turnover1993  >= 500000
    AND  C3.turnover1994  >= 500000;
```

```
INSERT INTO C1 (custkey, year, turnover)
  SELECT lo.custkey, od.orderyearkey, SUM(lo.endprice)
  FROM   lineitem_orders lo, orderday od
  WHERE  od.orderdate = lo.orderdate
    AND  od.orderyearkey IN (1992, 1993, 1994)
  GROUP BY lo.custkey, od.orderyearkey

  HAVING SUM(lo.endprice)  >= 500000;

INSERT INTO C2 (custkey)
  SELECT C1.custkey
  FROM   C1
  GROUP BY C1.custkey
  HAVING COUNT(*) = 3;

INSERT INTO C4 (custkey, turnover1992, turnover1993,
                turnover1994)
  SELECT C2.custkey, C1a.turnover, C1b.turnover, C1c.turnover
  FROM   C1 C1a, C1 C1b, C1 C1c, C2
  WHERE  C1a.custkey = C1b.custkey
    AND C1b.custkey = C1c.custkey
    AND C1c.custkey = C2.custkey
    AND C1a.year = 1992
    AND C1b.year = 1993
    AND C1c.year = 1994;

... See QS + MergeSelect
```

Figure 8. Modified Sequence for Request C (WhereToGroup)

4.2. Parallelism

In this section we discuss how parallel execution can support the query sequences generated by decision support applications. Our focus is not on intra-query parallelism. We assume that the DWDBS is able to generate parallel query plans for all queries in a sequence, where suitable.

Apart from this perspective on parallelism, it is also interesting to see whether the DSS optimizer can figure out a group of queries in the sequence that could be run in parallel. Therefore, our focal point is inter-query parallelism. For each sequence of queries, we can determine how the steps of the sequence depend on each other and describe these dependencies for example by means of a graph. The dependency graphs for the query sequences A and C of our application scenario are shown in Figure 9. For sequence A the graph shows, that *A1* and *A2* could be generated in parallel. For sequence C, the tables *C1*, *C2* and *C3* could first be generated in parallel and later in the sequence the same holds for tables *C5* and *C6*. In general, a group of queries can be executed in parallel, if none of the queries within the group depends on each other.

The dependency graph for a given sequence of queries can easily be determined by observing the FROM clauses of all queries. The graph consists of the temporary tables $Temp_i$ as nodes. An edge from node $Temp_i$ to $Temp_j$ exists if the FROM clause of query Q_j contains $Temp_i$. In particular, no knowledge about the application is required. Hence, this optimization strategy turns out to be applicable for the DSS optimizer.

The implementation of this strategy can be based on methods that generate parallel execution plans for SQL queries as they are used in standard database systems. These techniques produce an internal representation of the queries, which is extended by data and pipeline parallelism. The same methods could be used to exploit dependency graphs for query sequences like those in Figure 9. That is why available technology is very well suited for the DSS optimizer [7] [21].

But how about the interface of the DSS optimizer to the database system? The standard interface of a DWDBS allows the application to send one query after the other within one database transaction. For each of the SQL statements to be run in parallel, a separate connection and a separate transaction is necessary. Since those queries that are independent of each other should be started in parallel, only their access to the data warehouse base tables could inhibit the use of parallel transactions. If we assume that decision support applications only *read* base tables in the data warehouse, the standard SQL interface of database systems turns out to be sufficient for the optimization strategy we have discussed here.

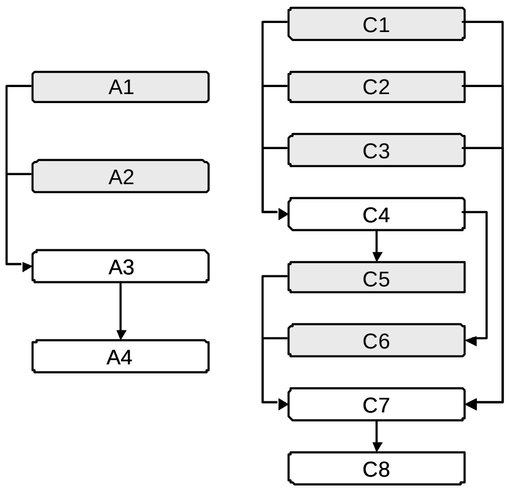


Figure 9. Dependency Graphs

4.3. Statistics

Query sequences generated by decision support applications usually produce the result data set that is relevant for the end user as the last step of the sequence. As soon as this last query terminates the final result is available and all other temporary tables could be dropped. For query sequence A in our example, table *A4* contains all relevant data for the end user whereas tables *A1*, *A2* and *A3* may be deleted as soon as *A4* is complete.

Each query in a sequence is processed by the optimizer of the DWDBS. One important factor that determines the efficiency of query execution is whether all necessary statistical information is available for the optimizer or not. Relevant statistical information is the number of rows, the number and type of all columns and the distribution of values in all columns of each table that is read by a query. Most database systems do not automatically update statistical information. For example, a separate statement is available for Oracle8*i* that starts the update of statistical information for a single table [22]. The following statement would be necessary to generate statistics for *A1*: `ANALYZE TABLE A1 COMPUTE STATISTICS`. Additional parameters may be used in order to make available more detailed information on the table and its indexes.

We assume that statistical information is up to date for all base tables of the data warehouse. However, this is not true for temporary tables generated by a sequence of queries. If the execution of a query sequence should be based on current statistics, the statements that provide this information have to be added to the sequence. The sequence for query C, as given in Figure 9, could be augmented by calling `ANALYZE TABLE` for the temporary tables *C1* through *C7*. The statistical information has to be available for each of these tables before the tables are read by another query in the sequence for the first time. The statistical information for table *C1* does not have to be present until the query for *C4* starts execution.

This leads to an additional strategy for the DSS optimizer that is applicable without any knowledge about the application that generated a sequence of queries. A sequence could be extended by statements that provide additional statistical information for the DWDBS optimizer. For this task the DSS optimizer has to know, how this is done for different database systems and which parameters influence the available statistics. It also has to take into account the trade-off between the time that is needed for the update of statistical information and the query execution time that is saved with execution plans based on this information. Further enhancements could be achieved by running statistic updates in parallel to other statements in the sequence. This is always possible if the queries do not depend on the temporary table for which the statistic update is in progress.

4.4. Partitioning and Indexing

All optimization strategies we have discussed so far are based on rewriting a sequence of SQL statements or enhancing the way it is executed. In this section, we will briefly discuss some approaches to support OLAP applications by changing the schema of the data warehouse.

Additional indexes on base tables in the data warehouse enable the optimizer of the DWDBS to find query plans that are more efficient. No rewriting of queries is required. It is necessary to find the proper combination of indexes that offer maximum support for all applications on the data warehouse and take into account given constraints on disk space and time for index maintenance. If typical query sequences on the data warehouse are known, administration tools of current database systems can be used to establish the suitable set of indexes [5] [13].

Partition tables support typical OLAP queries because they reduce the amount of base data that has to be scanned for one sequence of queries. One large fact table could include data for several years. If the typical query retrieves data for exactly one year, it might be more efficient to store the facts in partition tables for each year. The scan of one partition table instead of scanning the complete large fact table might then be sufficient for many queries. A smaller volume of data is likely to result in a more efficient retrieval of results. This kind of partitioning is supported by some OLAP tools [16]. The information how data is partitioned must be part of the meta data in order to make this strategy applicable for the query generator of the application.

Applications on a data warehouse can also be supported by aggregate tables. Especially OLAP applications require a lot of grouping and aggregation. If the aggregated data is already present in the data warehouse, queries run more efficient because less data has to be processed and less aggregations have to be computed. A base fact table, for example, could include data for each day, whereas most queries need data on a monthly or yearly basis. In this case, redundant aggregate tables should contain pre-aggregated data for each month and for each year. Some algorithms to find the proper set of aggregate tables are described in [12]. [32] describes a couple of approaches how a given query can be processed based on existing aggregate tables.

Our experimental results indicate that indexes, aggregate and partition tables are appropriate means to enhance query sequences. The performance enhancement is likely to decrease with the length of a query sequence. This is because in general only the statements at the beginning of a sequence access the base tables whereas the statements towards the end of the sequence mainly access temporary tables. Hence, the strategies mentioned here primarily support the statements at the beginning of query sequences.

Nevertheless, the DSS optimizer is not the suitable component to create indexes, aggregate tables or partition tables. The successful exploitation of these three strategies is based on knowledge about the schema of the data warehouse and the characteristics of all applications that access data in the warehouse. There is another important difference to the optimization approaches described in Section 4.1 through 4.3. All strategies we described there offer a local optimization for exactly one sequence of queries, whereas additional indexes and additional tables also influence the performance of other queries. Therefore, they should be part of the DWDBS administration.

5. Experimental Results

For our experiments, we have created the modified schema as described in Section 2.1 in a database system. We populated the tables with data based on the original TPC-H data, generated with three different scale factors according to the benchmark specifications. Our largest data set for which the results are presented here was produced with scale factor 10. The raw data summarizes to 10 GB, the main fact table *lineitem_orders* has about 60 million rows. The environment used for the experiments consists of a Sun Enterprise E4500 with 12 processors, 12 GB main memory, the object-relational database system IBM DB2/UDB V7.1 and a benchmark tool that is provided with DB2.

From a technological point of view, we could have used any other market-strength DBMS since we only used techniques that are commonly available (e.g. parallelism, indexes, partitioning as well as statistics). The decision for DB2 was twofold: First, we wanted to collect detailed information on the generated query plans. This information is provided by the EXPLAIN utility of DB2 [13]. Second, DB2 is known for its good optimization technology and thus can play a good reference point.

The results given in Table 2 are a subset of more than 400 experiments, where one experiment is the execution of one query sequence for one information request on one test data set. Each value presented here, is the average of at least three experiments. Some cells of Table 2 are empty because not all optimization strategies were applicable to all of the three information requests. Most of them are applicable to information request A and request C. Business questions that are similar to these two are very likely to build the majority of questions in a real environment.

Table 2. Experimental Results (in seconds)

	Optimization Strategy	Information Requests		
		A	B	C
1	QS	4877	498	30209
2	QS + MergeSelect	-	-	4188
3	QS + WhereToGroup	-	-	2296
4	SQ	2834	-	10734
5	SQ + MergeSelect	-	-	3853
6	SQ + WhereToGroup	-	-	4900
7	QS + Statistics	3109	-	5263
8	QS + Parallelism	3939	-	29004
9	QS + Indexes	5428	25	-
10	QS + Partitions	3712	26	27984

Looking at the results in more details, one can see that in almost all cases the optimization strategies were successful in reducing the runtime of the information requests. Sometimes there was only a slight enhancement whereas other strategies led to an execution time that was an order of a magnitude shorter than for the original query.

The execution time for the original sequence of queries (QS) is given in line 1. The modified versions of the query sequences and their combination into a single-query (SQ) as described in Section 4.1 are shown in lines 2 through 6. These results show that combining the statements of a sequence is likely to improve the performance. However, this is not true for information request C. The sequence *QS+WhereToGroup* runs much faster than the corresponding single-query. Hence, the DSS optimizer has to decide in which situation it is appropriate to combine the queries of a sequence and in which the eventually modified version of the sequence is the best choice.

Line 7 of Table 2 shows the results for the query sequence that was enhanced by generating additional statistical information for temporary tables. In our experiments, this version of the queries runs always faster than the original sequence. For information request C it was also faster than the very complex single-query (SQ), whereas for request A it was slower than the single-query. Therefore, leaving the query sequence as is and generating additional statistical information for temporary tables is a supplementary alternative to the other strategies, which can be used by the DSS optimizer.

The third basic alternative for the DSS optimizer is to run the sequence of queries as it was generated, but with some queries of the sequence in parallel as described in Section 4.2. The experimental results for this strategy are given in line 8. They show only a slight improvement compared to the original sequence. Especially for information request C the enhancement is less than

10 percent. This is because the runtime of the query sequence for this question is dominated by only one query that joins several temporary tables.

Some results for the strategies that turned out not to be appropriate for the DSS optimizer are given in lines 9 and 10. With additional indexes and partitions, we can see a remarkable performance enhancement for information request B. Additional partition tables reduced the runtime for information requests A and C, where the reduction is between 8 and 25 percent. For the information requests A and C we achieved further improvements by other strategies that are useable by the DSS optimizer. Hence, we do not loose too much optimization potential if the DSS optimizer has to ignore additional indexes and partitions as optimization strategies. Nevertheless, in a real environment these strategies are mandatory for the administrator of the data warehouse.

6. Conclusion

In this work we investigated that an optimizer in between decision support applications and the DWDBS is essential for the efficient processing of some classes of typical information requests generated by DSS tools. Hence, we suggested an optimized system architecture for decision support applications. In this architecture, generated query sequences are rewritten by an additional system component, called DSS optimizer. The transformation is based on optimization strategies that are independent of the application and use little knowledge about the underlying database system. This DSS optimizer has two main advantages. First, it is able to support several applications without any need to change their query generation algorithms. Second, it is capable of applying optimization strategies that are not supported by state-of-the-art database systems.

We set up a realistic application scenario, selected a set of relevant information requests and ran a large series of experiments based on TPC-H data. Our experiments have shown that all three strategies we proposed for the DSS optimizer were successful. They reduced the runtime of the given query sequences significantly. None of the three strategies turned out to be the best in all situations. Hence, it is the task of the DSS optimizer to decide upon the strategy to use. This decision should be based on information gained from the query sequence itself and on meta data gained from the underlying database system. Therefore, further work will focus on heuristics that could be used by the DSS optimizer in order to decide which optimization strategy or which combination of strategies should be used for a given sequence of queries.

Another important issue of future work is the design and implementation of the DSS optimizer. The basic technology for a DSS optimizer is in place because all three strategies are based on matured technology. Generating single-queries is similar to view expansion. Determining queries within a sequence that could run in parallel is based on parallel database technology [7] [21]. Using statistical information for query optimization is a basic optimization technology. Furthermore, we want to apply extensible optimization technology as is available with systems like CASCADES [8]. In doing so, one has to observe that the DSS optimizer is not a general-purpose optimizer, but only a specific component that supports only a predefined set of optimization strategies. Consequently, we want to reconsider the main design decisions of an optimizer e.g. search space, rule set and cost-based decision making. As a result, we want to come up with a tailored and efficient DSS optimization component.

Acknowledgement. We would like to thank Leonard Shapiro and Ralf Rantzau for helpful discussions and their comments on an early version of the paper.

References

[1] AberdeenGroup: Bringing Analytical Reporting to Enterprise Business Intelligence. Aberdeen Group, Boston, 1999.

[2] R. Agrawal, J. C. Shafer: Parallel Mining of Association Rules. In: TKDE 8(6), 1996.

[3] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, K. Shim: Optimizing Queries with Materialized Views. In: ICDE, March 1995.

[4] S. Chaudhuri, U. Dayal: An Overview of Data Warehousing and OLAP Technology. In: SIGMOD Record, Vol. 26., No. 1, 1997.

[5] S. Chaudhuri, V. Narasayya: AutoAdmin "What-if" Index Analysis Utility. In: SIGMOD Record, Vol. 27, No. 2, 1998.

[6] C. J. Date, H. Darwen: A guide to the SQL standard. 4th ed., Addison-Wesley, Reading, 1997.

[7] D. DeWitt, J. Gray: Parallel Database Systems: The Future of High Performance Database Systems. In: CACM, Vol. 35, No. 6, pp. 85-92, 1992.

[8] G. Graefe: The Cascades Framework for Query Optimization. In: DE Bulletin, Vol. 18, No. 3, 1995.

[9] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, H. Pirahesh: Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab and Sub-Totals. In: Data Mining and Knowledge Discovery, Vol. 1, No. 1, 1997.

[10] P. Gulutzan, Trudy Pelzer: SQL-99 Complete, Really. R&D Books, Lawrence, 1999.

[11] J. Han, M. Kamber: Data Mining: Concepts and Techniques. Morgan Kaufmann, 2000.

[12] V. Harinarayan, A. Rajaraman, J. D. Ullman: Implementing Data Cubes Efficiently. In: SIGMOD Record, Vol. 25, No. 2, 1996.

[13] IBM: DB2 Command Reference, 1999.

[14] Informix: Data Warehousing for the Retail Industry. White Paper, http://www.informix.com, 1998.

[15] R. Kimball: The Data Warehouse Toolkit. John Wiley & Sons, New York, 1996.

[16] MicroStrategy: The Case for Relational OLAP. White Paper, MicroStrategy, 1995.

[17] B. Mitschang: Query Processing in Database Systems (in German), Vieweg-Verlag, 1995.

[18] P. O'Neil, G. Graefe: Multi-Table Joins Through Bitmapped Join Indices. In: SIGMOD Record, Vol. 24, No. 3, 1995.

[19] P. O'Neil, D. Quantas: Improved Query Performance with Variant Indexes. In: SIGMOD Record Vol. 26, No. 2, 1997.

[20] C. Nippl: Providing Efficient, Extensible and Adaptive Intra-query Parallelism for Advanced Applications. Technische Universität München, Dissertation, 2000.

[21] C. Nippl, B. Mitschang: TOPAZ: A Cost-Based, Rule-Driven, Multi-Phase Parallelizer. Proc. VLDB Conf., New York, 1998.

[22] Oracle Corporation: Oracle8*i* SQL Reference. Release 3 (8.1.7), Oracle, September 2000.

[23] D. Peppers, M. Rogers: Data Warehousing and Retailing. DM Reviews, October 1998.

[24] SAS Institute: Finding the Solution to Data Mining. White Paper, SAS Institute, 1998.

[25] A. Sen, V. S. Jacob: Industrial-Strength Data Warehousing. In CACM Vol. 41, No. 9, 1998.

[26] D. Srivastava, S. Dar, S. Jagadish, A. Levy. In: VLDB, September 1996.

[27] S. N. Subramanian, S. Venkataraman: Cost-Based Optimization of Decision Support Queries using Transient-Views. In: SIGMOD Record, Vol. 27, No. 2.

[28] Thinking Maschines Corporation: Darwin Reference. Release 3.0.1. Thinking Maschine Corporation, 1998.

[29] D. S. Tkach: Information Mining with the IBM Intelligent Miner Family. White Paper, IBM, 1998.

[30] Transaction Processing Performance Council: TPC Benchmark H (Decision Support) Standard Specification, Revision 1.1.0, June 1999.
[31] R. Wagner: Optimization of an OLAP Application for Retailers (in German). University of Stuttgart, Faculty of Computer Science, Project Paper, Nr. 1770, 2000.
[32] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, M. Urata: Answering Complex SQL queries Using Automatic Summary Tables. In: SIGMOD Record, Vol. 29, No. 2, 2000.
[33] Y. Zhao, P. M. Deshpande, J. F. Naughton, A. Shukla: Simultanous Optimization and Evaluation of Multiple Dimensional Queries. In. SIGMOD Record, Vol. 27, No. 2, 1998.

Appendix

```
INSERT INTO B1 (partkey, partname, custregionkey, custregionname, custnationkey,
                custnationname, orderyearkey, orderyear, sumquantity)
  SELECT pa.partkey, pa.partname, cr.custregionkey, cr.custregionname, cn.custnationkey,
         cn.custnationname, oy.orderyearkey, oy.orderyear, SUM(lo.quantity)
  FROM   lineitem_orders lo, customer cu, orderday od, part pa, custregion cr, custnation cn,
         orderyear oy
  WHERE  cu.custkey = lo.custkey AND od.orderdate = lo.orderdate
    AND  lo.partkey = pa.partkey AND cu.custregionkey = a5.custregionkey
    AND  cu.custnationkey = cn.custnationkey AND  od.orderyearkey = oy.orderyearkey
    AND  pa.partkey <= 3 AND  oy.orderyearkey = 1994
  GROUP BY pa.partkey, cr.custregionkey, cn.custnationkey, oy.orderyearkey;
```

Figure 10. Query Sequence for Request B

Q1

```
INSERT INTO C1 (custkey, turnover1992)
  SELECT lo.custkey, SUM(lo.endprice)
  FROM   lineitem_orders lo, orderday od
  WHERE  od.orderdate = lo.orderdate
    AND  od.orderyearkey = 1992
  GROUP BY lo.custkey;
```

Q2

```
INSERT INTO C2 (custkey, turnover1993)
  SELECT lo.custkey, SUM(lo.endprice)
  FROM   lineitem_orders lo, orderday od
  WHERE  od.orderdate = lo.orderdate
    AND  od.orderyearkey = 1993
  GROUP BY lo.custkey;
```

Q3

```
INSERT INTO C3 (custkey, turnover1994)
  SELECT lo.custkey, SUM(lo.endprice)
  FROM   lineitem_orders lo, orderday od
  WHERE  od.orderdate = lo.orderdate
    AND  od.orderyearkey = 1994
  GROUP BY lo.custkey;
```

Q4

```
INSERT INTO C4 (custkey, turnover1992, turnover1993,
                turnover1994)
  SELECT C1.custkey, C1.turnover1992, C2.turnover1993,
         C3.turnover1994
  FROM   C1, C2, C3
  WHERE  C1.custkey = C2.custkey
    AND  C1.custkey = C3.custkey
    AND  C1.turnover1992 >= 500000
    AND  C2.turnover1993 >= 500000
    AND  C3.turnover1994 >= 500000;
```

Q5

```
INSERT INTO C5 (custkey, stddeviation)
  SELECT lo.custkey, STDDEV(lo.endprice)
  FROM   lineitem_orders lo, orderday od, C4
  WHERE  od.orderdate = lo.orderdate
    AND  od.orderyearkey IN (1992, 1993, 1994)
    AND  lo.custkey = C4.custkey
  GROUP BY lo.custkey;
```

Q6

```
INSERT INTO C6 (custkey, stddeviation)
  SELECT lo.custkey, STDDEV(lo.endprice) / AVG(lo.endprice)
  FROM   lineitem_orders lo, orderday od, C4
  WHERE  od.orderdate = lo.orderdate
    AND  od.orderyearkey IN (1992, 1993, 1994)
    AND  lo.custkey = C4.custkey
  GROUP BY lo.custkey;
```

Q7

```
INSERT INTO C7 (custkey, custname, stddev1, stddev2,
                turnover1992, turnover1993, turnover1994)
  SELECT cu.custkey, cu.custname, C5.stddeviation,
         C6.stddeviation, C1.turnover1992,
         C2.turnover1993, C3.turnover1994
  FROM   C1, C2, C3, C5, C6, customer cu
  WHERE  C5.custkey = C1.custkey
    AND  C5.custkey = C2.custkey
    AND  C5.custkey = C3.custkey
    AND  C5.custkey = C6.custkey
    AND  C5.custkey = cu.custkey;
```

Q8

```
INSERT INTO C8 (custkey, custname, stddev1, stddev2,
                turnover1992, turnover1993, turnover1994)
  SELECT C7.custkey, C7.custname, C7.stddev1,
         C7.stddev2, C7.turnover1992,
         C7.turnover1993, C7.turnover1994
  FROM   C7
  WHERE  C7.stddev2 <= 0.66794004454646;
```

Figure 11. Query Sequence for Request C

```
INSERT INTO C8 (custkey, custname, stddev1, stddev2, turnover1992, turnover1993, turnover1994)
  SELECT C7.custkey, C7.custname, C7.stddev1, C7.stddev2, C7.turnover1992, C7.turnover1993,
         C7.turnover1994
  FROM
       (SELECT cu.custkey, cu.custname, C5.stddeviation, C6.stddeviation,
               C1.turnover1992, C2.turnover1993, C3.turnover1994
        FROM
            (SELECT lo.custkey, SUM(lo.endprice)
             FROM   lineitem_orders lo, orderday od
             WHERE  od.orderdate = lo.orderdate AND od.orderyearkey = 1992
             GROUP BY lo.custkey
            ) AS C1 (custkey, turnover1992),
            (SELECT lo.custkey, SUM(lo.endprice)
             FROM   lineitem_orders lo, orderday od
             WHERE  od.orderdate = lo.orderdate AND od.orderyearkey = 1993
             GROUP BY lo.custkey
            ) AS C2 (custkey, turnover1993),
            (SELECT lo.custkey, SUM(lo.endprice)
             FROM   lineitem_orders lo, orderday od
             WHERE  od.orderdate = lo.orderdate AND od.orderyearkey = 1994
             GROUP BY lo.custkey
            ) AS C3 (custkey, turnover1994),
            (SELECT lo.custkey, STDDEV(lo.endprice)
             FROM lineitem_orders lo, orderday od,
                  (SELECT C1.custkey, C1.turnover1992, C2.turnover1993, C3.turnover1994
                   FROM
                        (SELECT lo.custkey, SUM(lo.endprice)
                         FROM   lineitem_orders lo, lookup_orderday od
                         WHERE  od.orderdate = lo.orderdate AND od.orderyearkey = 1992
                         GROUP BY lo.custkey
                        ) AS C1 (custkey, turnover1992),
                        (SELECT lo.custkey, SUM(lo.endprice)
                         FROM   lineitem_orders lo, orderday od
                         WHERE  od.orderdate = lo.orderdate AND od.orderyearkey = 1993
                         GROUP BY lo.custkey
                        ) AS C2 (custkey, turnover1993),
                        (SELECT lo.custkey, SUM(lo.endprice)
                         FROM   lineitem_orders lo, orderday od
                         WHERE  od.orderdate = lo.orderdate AND od.orderyearkey = 1994
                         GROUP BY lo.custkey
                        ) AS C3 (custkey, turnover1994)
                   WHERE C1.custkey = C2.custkey AND C1.custkey = C3.custkey
                     AND C1.turnover1992 >= 500000 AND C2.turnover1993 >= 500000
                     AND C3.turnover1994 >= 500000
                  ) AS C4 (custkey, turnover1992, turnover1993, turnover1994)
             WHERE od.orderdate = lo.orderdate AND od.orderyearkey IN (1992, 1993, 1994)
                   AND lo.custkey = C4.custkey
             GROUP BY lo.custkey
            ) AS C5 (custkey, stddeviation),
      (SELECT lo.custkey, STDDEV(lo.endprice) / AVG(lo.endprice)
             FROM lineitem_orders lo, orderday od,
                  (SELECT C1.custkey, C1.turnover1992, C2.turnover1993, C3.turnover1994
                   FROM
                        (SELECT lo.custkey, SUM(lo.endprice)
                         FROM   lineitem_orders lo, orderday od
                         WHERE  od.orderdate = lo.orderdate AND od.orderyearkey = 1992
                         GROUP BY lo.custkey
                        ) AS C1 (custkey, turnover1992),
                        (SELECT lo.custkey, SUM(lo.endprice)
                         FROM   lineitem_orders lo, orderday od
                         WHERE  od.orderdate = lo.orderdate AND od.orderyearkey = 1993
                         GROUP BY lo.custkey
                        ) AS C2 (custkey, turnover1993),
                        (SELECT lo.custkey, SUM(lo.endprice)
                         FROM   lineitem_orders lo, lookup_orderday od
                         WHERE  od.orderdate = lo.orderdate AND od.orderyearkey = 1994
                         GROUP BY lo.custkey
                        ) AS C3 (custkey, turnover1994)
                   WHERE C1.custkey = C2.custkey AND C1.custkey = C3.custkey
                     AND C1.turnover1992 >= 500000 AND C2.turnover1993 >= 500000
                     AND C3.turnover1994 >= 500000
                  ) AS C4 (custkey, turnover1992, turnover1993, turnover1994)
             WHERE od.orderdate = lo.orderdate AND od.orderyearkey IN (1992, 1993, 1994)
                   AND lo.custkey = C4.custkey
             GROUP BY lo.custkey
            ) AS C6 (custkey, stddeviation),
            customer cu
       WHERE C5.custkey = C1.custkey AND C5.custkey = C2.custkey AND C5.custkey = C3.custkey
         AND C5.custkey = C6.custkey AND C5.custkey = cu.custkey
     ) AS C7 (custkey, custname, stddev1, stddev2, turnover1992, turnover1993, turnover1994)
  WHERE  C7.stddev2 <= 0.66794004454646;
```

Figure 12. Single Query for Request C