Cost-Based Optimization of Decision Support Queries using Transient-Views

Subbu N. Subramanian*
IBM Santa Teresa Labs
San Jose, CA 95141

Shivakumar Venkataraman
IBM Santa Teresa Labs
San Jose, CA 95141

Abstract

Next generation decision support applications, besides being capable of processing huge amounts of data, require the ability to integrate and reason over data from multiple, heterogeneous data sources. Often, these data sources differ in a variety of aspects such as their data models, the query languages they support, and their network protocols. Also, typically they are spread over a wide geographical area. The cost of processing decision support queries in such a setting is quite high. However, processing these queries often involves redundancies such as repeated access of same data source and multiple execution of similar processing sequences. Minimizing these redundancies would significantly reduce the query processing cost. In this paper, we (1) propose an architecture for processing complex decision support queries involving multiple, heterogeneous data sources; (2) introduce the notion of *transient-views* – materialized views that exist only in the context of execution of a query – that is useful for minimizing the redundancies involved in the execution of these queries; (3) develop a *cost-based* algorithm that takes a query plan as input and generates an optimal "covering plan", by minimizing redundancies in the original plan; (4) validate our approach by means of an implementation of the algorithms and a detailed performance study based on TPC-D benchmark queries on a commercial database system; and finally, (5) compare and contrast our approach with work in related areas, in particular, the areas of answering queries using views and optimization using common sub-expressions. Our experiments demonstrate the practicality and usefulness of transient-views in significantly improving the performance of decision support queries.

1 Introduction

Recent advances in database research is paving the way for seamless access to electronic data available in a variety of sources including traditional database systems, repositories on the internet/World Wide Web, semi-structured documents and file systems. Indeed, the ability to access, integrate, and reason over data from diverse data sources is critical for many next generation database applications. This 'interoperability' problem has recently received considerable attention from database researchers [ASD+91, CGH+94, CAS94, PGW95, SAB+95, TRV96, LSS96, AMMT96, Abi97, HKWY97], to name a few). The majority of the approaches are based on the idea of developing a database-like 'wrapper' for data sources and implementing queries against these sources [CGH+94, HKWY97, TRS97]. Typically, wrappers provide a relational or object-relational view of the data in the non-traditional sources, and enable the user to use a common language/interface to query data from the diverse sources. Systems that provide end users with an integrated view of data in multiple data sources are referred to as Heterogeneous Database Systems (HDBS) and Multi-database Systems (MDBS), and are increasingly becoming relevant in the context of real-life business applications.

To illustrate, consider an application where an investment broker manages investment portfolios of his clients. The portfolio information is stored in a relational database, which also contains other information about the clients, such as, their address, profession, and etc. The broker obtains the latest as well as historical stock price information from the stock exchange servers on the Web, and maintains the account information of the clients in a spreadsheet. In order to make complex decisions involving the buying and selling of stocks for the clients, the broker would have to analyze and compare information from all of these sources.

Typically, these complex queries contain sub-queries that share computational steps. Analysis of the TPCD [TPC93] benchmark queries reveals that there is redundancy in computation even in simple queries Currently available database query optimizers do not have the wherewithal to identify these redundancies so that the results of one execution could be used for processing other segments of the query. *Since decision support queries are typically time consuming to run, especially in a HDBS setting, identifying and sharing computational results judiciously, could lead to significant improvements in performance.* The example in the following section illustrates the kind of redundant computation that is typical of decision support queries.

1.1 Motivating Example

Consider the following decision support query, the investment broker of the previous section wishes to express: *List technology stocks owned by computer engineers that have a higher average sales volume over the past year, than the maximum sales volume any oil stock owned by a chemical*

*On assignment from IBM Almaden Research, San Jose, CA

SIGMOD '98 Seattle, WA, USA

engineer reached in the first six months of the year; also list the name of the computer engineer.

If we assume a relational wrapper for the data sources, and hence an SQL interface, the following is the query, the broker would express.

```
DB2:Rinvest(name, prof, ticker, qty, date, price)
Web: Wstock(ticker, cat, date, vol, endprice)

SELECT Rinvest.name, Rinvest.ticker
FROM   Rinvest, Wstock
WHERE  Wstock.cat    = 'Tech' AND
       Rinvest.Profession= 'Computer Engineer'
       Wstock.date <= '12/31/97' AND
       Wstock.date >= '01/01/97' AND
       Rinvest.ticker = Wstock.ticker
GROUP BY Rinvest.name,Rinvest.ticker,Wstock.ticker
HAVING   AVG(Wstock.vol) >
   (  SELECT MAX(Wstock.vol)
      FROM   Wstock, Rinvest
      WHERE  Rinvest.cat  = 'Oil'
        AND  Rinvest.Profession='Chemical Engineer'
        AND  Rinvest.ticker=Wstock.ticker
        AND  Wstock.date   <= '06/30/97'
             Wstock.date   >= '01/01/97' )
```

The above query contains two query blocks: The first query block computes the average sales volume per day, of the technology stocks owned by computer engineers over the year 1997. The second query block computes the maximum sales volume of any oil stock owned by a chemical engineer reached in the first six months of 1997.

Operationally, one way of processing this query would involve the following steps. These steps closely reflect the way most relational query optimizers execute the above query.

1. query the web data source to identify the subrelation of oil stocks in the Wstock source for the first six months of 1997;
2. obtain portfolio information for investors who are chemical engineers;
3. join portfolio information with the volume information for oil stocks obtained from the web;
4. perform a group by on volume to obtain the maximum volume of sales any oil stock traded;
5. query the web source to identify the tech stocks in the Wstock source for 1997;
6. obtain portfolio information for investors who are computer engineers;
7. join the investor portfolio information with the volume information of tech stocks;
8. perform a group by on the ticker to obtain the average volume of sales for each tech stock for each investor;
9. filter the tech stocks based on if the average volume exceeds the number obtained in step (4);

Note the similarity between step (1) and step (5) – both of them connect to the web source and obtain information on a certain type of stock. Since connecting multiple times to the web source is detrimental to performance, it would be better to combine these steps together – connect to the web source once and issue a query that gets both the oil stocks information *and* the technology stocks information at the same time, and later perform the necessary filtering operations to separate the oil data from the technology data. This would of course incur the extra overhead of storing the 'temporary data' and the cost of applying the filtering operations. However, the point to be noted is that a worthwhile trade-off exists between these two ways of performing the steps. Similarly, we can combine steps (2) and (6) to obtain with one scan of the Rinvest table, the portfolio information of investors who are chemical engineers or who are computer engineers. This would especially be profitable in the event that the Rinvest table does not have any index. Also, we can combine steps (3) and (7) to perform the join between the combined information obtained from the web source and from the investor table, in one shot. The group by operation obtained by combining (3) and (7) can also be combined into a single operation. This result can then be used to compute the average volume of technology stocks and the maximum volume of the oil stocks.

If we were to exploit the 'redundancies' discussed above, the following would be the steps involved in processing the query.

1. query the web data source to identify the subrelation of oil stocks and tech stocks at the web source for the year 1997; store the results in a temporary relation.
2. obtain portfolio information for investors who are chemical engineers or computer engineers; store the results in a temporary relation.
3. join the volume information obtained from the web (temporary relation of step (1)) with the portfolio information (temporary relation of step (2)).
4. perform a group by on investor name, investor profession, and volume to obtain the maximum volume, and average volume of sales for the technology and oil stocks and store this in a temporary relation.
5. scan the temporary relation of step (4) to obtain the maximum volume for the oil stocks owned by chemical engineers.
6. scan the temporary relation of step (4) to obtain the average volume of the tech stocks owned by computer engineers.
7. filter the tech stocks based on if the average volume exceeded the number obtained in step (5).

This example illustrates that various steps in the processing of even some simple decision support queries. Some of these steps can be combined and later compensated with an aim towards improving overall performance. Our work was inspired by this observation. Clearly it is *not* certain that such a combine and compensate approach would lead to an improvement in performance. In this paper, we address the following questions: *How can we exploit the phenomenon of repeated access of same data sources, and multiple execution of similar processing sequences, inherent in processing complex decision support queries? What are the conditions under which combining steps and compensating for them later would lead to an improvement in performance? Which stage of query processing should we perform this analysis? And, how do we develop the necessary algorithms that are efficient, and at the same time implementable in commercial query processing systems in a non-intrusive manner?*

1.2 Our Approach and Summary of Contributions

In this paper, we study the problem of exploiting similarities in the steps involved in processing a complex decision support query. Towards this end, we introduce a notion of *transient-views* – materialized views that exist only in the context of execution of a query. Transient-views are an encapsulation of similar execution steps combined into one. Our approach is based on analyzing the query plan generated by an optimizer, for 'equivalent' subparts in the plan. We propose a notion of equivalent plans that is quite broad and hence allows for combining a large number of processing steps. We develop efficient algorithms for identifying similar plans, combining them into transient-views, and filtering the views for compensation. The plan operators we consider include the SCAN , JOIN , GROUP BY , UNION operators as well as a novel RQUERY (Remote Query) operator. This latter operator assumes significance especially in the context of HDBS systems, where queries get executed in remote sources and the data returned is shipped over the wire, in the process incurring heavy connection and network costs. Also, the fact that our approach does not interfere with the functions of the query optimizer, makes it especially suitable in the context of HDBS systems. This is due to the fact that in these systems, the join enumerations of the query optimizer play a crucial role in improving performance. Our approach is cost-based and bootstraps on existing costing infrastructure. A cost-based approach allows us to rigorously study the scenarios in which applying our technique of transient-view materialization would lead to a definite improvement in performance. The algorithms we propose, require simple extensions to the query optimizer. We demonstrate the ease of implementing the algorithms by implementing and realizing it in the context of a commercial HDBS query optimizer. We also undertake an extensive performance study based on the TPC-D benchmark queries and demonstrate that using *transient-views* leads to significant improvement in the performance of decision support queries.

1.3 Outline of the Paper

This paper is organized as follows. In Section 2 we discuss related work. We describe the query processing architecture in Section 3. In Section 4 we introduce the notion of equivalent plans and describe an algorithm to identify equivalent subplans in a plan tree. Following this, in Section 5 we develop algorithms that generate transient-views, based on the information about equivalent subplans. We also develop a cost model that aids in evaluating the cost of executing queries when the transient-views are materialized. In Section 6, we run several performance experiments based on the TPCD benchmark queries and present an analysis of our results. Finally, we discuss future research and conclude in Section 7.

2 Related Work

One of the early works on optimizing queries with common subexpression is the work of Hall [Hal76]. This work was based on identifying common subexpressions by examining the query syntax. Identifying redundant expressions based on the query syntax can have a detrimental effect of performance as it limits the optimizer's choice for generating optimal plans. We illustrate this using an example in Section 6. In contrast, our work has its basis on identifying query plans that are similar but not necessarily identical. Since we identify subplan execution that can be shared after optimization, our approach does not limit the optimizer choices. The cost-based approach that we use ensures that our substitutions only result in better plans. We compare transient-views with a syntax based approach in Section 6.

The work closely related to ours is that of Sellis [Sel88]. His work is in the context of optimizing multiple queries submitted as a batch of queries to the database. Sellis describes two algorithms, Interleaved Execution(IE) and Heuristic Algorithm (HA), to identify common subplans that are identical when multiple queries are submitted for batch execution. The IE algorithm generates several redundant temporary relations for each pair of identical subplan operators. The HA algorithm is based on the work by Grant and Minker [GM80] and [GM81] uses many non-optimal intermediate query plans for each query in the query batch to optimize the plan for the batch of queries so that the global plan is optimized. The HA and the IE algorithm require a strict notion of equality between query plans for them to be combined. The algorithms also assume that the identical subplans can be found easily. They also assume that the join predicates are simple equi-joins. In contrast, we use a very broad definition of equivalence, that includes plans with different predicates, different group by and join predicates.

In an attempt to find fast access paths for view processing, Roussopoulos [Rou82b] and [Rou82a] provides a framework for interquery analysis based on query graphs introduced by Wong and Youssefi ([WY76]). Kim [Kim84] describes a two stage optimization procedure similar to that of [GM80]. The unit of sharing in queries in Kim's proposal is a relation and does not extend to arbitrary expressions that are part of a query. Thus his work is applicable in a restricted setting of single relation queries.

There are several papers that are based on caching query results, for answering future queries. Franklin [DFJ+96] use semantic data caching to identify data in the client buffer in a client-server environment. Finkelstein [Fin82] also use similar methods to cache data of previously executed queries to answer future queries. Jarke [Jar84] discusses the problem of common subexpression isolation. He presents several different formulations of the problem under various query language frameworks. He discusses how common expressions can be detected and used according to their type.

Recently there has been a lot of work in the area of using materialized views to answer decision support queries. These papers are related to our work in that, transient-views are also materialized views used for improving efficiency of query processing. There is also research work that addresses the problem of maintaining materialized views so that they can be maintained efficiently when there are updates to the base tables. Ross et al. [RSS96] address the problem of efficiently maintaining materialized views by maintaining other materialized views. Srivastava et al. [SDJL96] and Levy et al. [LMSS95] describe algorithms to determine the portions of the query that can be expressed efficiently using the definition of materialized views. Chaudhuri et al. [CKPS95] describe the problem of optimizing queries in the presence of materialized views. They identify portions of the query that can be answered with the materialized view and determine if it is efficient to answer the query using the materialized view.

In the area of Online Analytical Processing (OLAP), papers have studied the problem of determining the views to materialize in the presence of space constraints so that computation can be speeded up to compute the Cube By operator proposed by Gray et al [GBLP95]. These papers are related to our work in that, they address the specific

problem of using materialized views while computing the Cube operator. Harinarayanan et al. [HRU96] have studied the problem of computing data cubes efficiently by materializing intermediate results. They have also studied algorithms on when to construct indexes on intermediate results [GHRU97]. Gupta [Gup97] develop a theoretical framework for identifying the views to materialize so that they can be maintained efficiently in the presence of storage constraints.

3 Architecture

In this section, we describe the HDBS architecture which forms the setting for the work described in this paper. This architecture is loosely based on the DataJoiner [VZ97] heterogeneous database system. The HDBMS provides a single database image to tables (or table views obtained via relational wrappers) in multiple data sources. The database clients access remote tables as local tables through user defined aliases. The presence of remote data sources is completely transparent to the user and the user submits queries as though the queries were directed to local tables and gets back results as though they would have been generated on local tables. Queries submitted to the HDBS may access both the local and remote tables. The HDBS query processor rewrites these queries, applies cost-based optimization to determine the which parts of the queries should be executed locally, and which should be 'pushed-down' to the remote sources. The parts of the query that should be executed remotely are translated to the lingua-franca (in the case of SQL databases, to the appropriate SQL dialects) of the remote sources and sent to the remote source for execution. The results received are translated, converted to local types, and combined with the results of queries on local tables and returned to the user.

3.1 Query Optimizer

Our optimizer architecture is based on the well-known STARBURST relational query optimizer [GLS93] and [HFLP89], extended to a HDBS setting. Figure 1 shows the stages that a query goes through in the optimizer before it is executed. Following is a brief description of each stage in the query optimizer:

- **Parser:** The query from the user is first parsed, semantically analyzed, and translated into a chosen internal representation that is referred to as the Query Graph Model (QGM).
- **Query Rewrite:** The output of the parser is transformed by a set of rewrite rules [PHH92]. These rewrite rules are usually heuristics aiding in transforming the query representation into a better form in order for the the query optimizer to search a larger space for an optimal plan.
- **Query Optimizer:** The optimizer uses the rewritten query as input to generate query plans based on the capabilities of the remote data source.
- **Code Generation:** Code is generated for those portions of the query that needs to be executed locally as well as remotely.

Figure 1 shows how our transient-view module fits in the above query processing architecture. The module takes as input the plan produced by the optimizer and identifies the

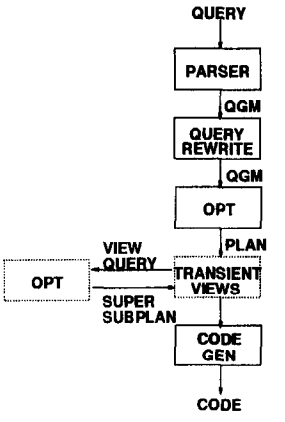

Figure 1: Optimizer Architecture

redundant computation in the plan. The optimizer plan is modified so that identical subplans are merged into a superplan. The superplan is generated by the optimizer, based on the plan properties of the subplans being merged. The plan that is fed into the code generator consists of a global plan with redundant subplans eliminated.

4 Equivalence Algorithm

In this section, we introduce the terminology used in the rest of the paper and our notion of equivalence of query plans. We also present an algorithm that helps identify equivalent subplans in a given query plan.

4.1 Definitions and Notation

Our approach to optimizing decision support queries is based on studying the query execution plan generated by a cost-based optimizer [HFLP89], [JV84], and [SAC+79]. Naturally, we make use of the *Plan tree* as the underlying data structure for representing the query execution plan.

Plan tree: The plan tree corresponding to a query Q is a directed acyclic graph (DAG) whose internal nodes correspond to the plan operators (introduced below) and the leaf nodes correspond to the relations at the data source. The plan "tree" could be a DAG since it could have common subexpressions. The algebraic expression computed by the root node of the tree is equivalent to Q.

Plan Properties: Associated with a plan is a set of properties that provide meta-information on the work done by the plan. Some of the properties naturally associated with a plan include: the tables involved in the execution of the plan, columns selected, aggregation functions, columns involved in the aggregation operation, the simple and join predicates, data sources where the plan executes, the estimated cost of executing the plan, and an estimate of the number of rows returned by the plan. The properties information summarizes the various tasks underlying the plan and its cost. Table 1 lists the properties we consider along with the notation we use for representing them [1].

Plan Operators: The plan operators are an encapsulated representation of the fundamental tasks performed by the

[1] In order to keep the description simple we have not included the order and key property in the property list. For the purposes of this paper, if two plans differ in the order and key property, they are will not be considered equivalent. Please refer to [SV98] we have defined a broader notion of plan equivalence to include the order and key properties

Property	Notation	Example
Tables	T_i	Relation T1, T2
Columns	$Cols_i$	C1, C2, C3
Join preds	JP_i	T1.C1 RelOp T2.C2
Simple preds	SP_i	T1.C1 RelOp Const
Aggregation	$AF_i(Exp)$	SUM, MIN, COUNT
Group by list	GC_i	GROUP BY C1, C2
Data Source	S_i	DB2, WEB, ORACLE
Cost	$Cost_i$	Time in ms
Cardinality	$Card_i$	Number of Tuples

Table 1: Plan Properties

Operator	Input Parameter
SCAN	Type (scan *vs* iscan) Name of Table Simple Predicates Columns selected
JOIN	Method (merge, nested ...) Outer Plan Inner Plan Join Predicates
GROUP BY	Columns in select Aggregation Functions Group By list
RQUERY	Input plan Data source

Table 2: Operator and their Parameters

query engine. The following is the list of operators [TRS97] we consider in our work: SCAN , JOIN , GROUP BY , and RQUERY . These operators form the building blocks for the query plan. We choose the above list of operators as they are sufficient to describe any plan tree with the properties described in Table 1 . The SCAN , JOIN , and the GROUP BY operators do not need any introduction. The remote query operator, RQUERY , is a novel operator that is unique to a setting consisting of remote data sources. It encapsulates the job (subplan) to be performed at a remote source. Thus, the unit of work 'beneath' this operator is performed at the remote source, and the data that is returned as a result of this execution is shipped to the local site.

Each operator can be thought of as taking a plan or a set of plans and their associated properties as input and yielding a plan with a modified set of properties as output. Thus, the properties of a plan is recursively built using the plan operators. Table 2 presents the operators and the input parameters associated with them.

To illustrate the above concepts, we now present an example of a query and its corresponding plan tree.

Example 4.1 Consider the following query in the context of our investor application of Section 1.1. *Display the investor names and their investment value in technology stocks and oil stocks as of '11/03/97' for those investors for whom the average trading volumes of the technology and oil stocks in their portfolio was higher than the market trading volume of oil and technology stocks. Consider only those stocks in the portfolio that were bought prior to '10/01/97'.*

The following is an SQL rendering of the above query. For the sake of clarity, rather than a compelling need, we express this query via view definitions.

```
/* This view retrieves the name, average volume,*/
/* number of stocks, and the net value of the   */
/* investors investments in technology stocks   */

CREATE VIEW Tech(name, id, vol, cnt,value)
AS SELECT   Rinvest.name, AVG(Wstock.vol),
            COUNT(Wstock.vol),
            SUM(Rinvest.qty*Wstock.price)
  FROM   Rinvest, Wstock
  WHERE  Rinvest.ticker=Wstock.ticker
    AND  Wstock.cat='Tech'
    AND  Wstock.date = '11/03/97'
    AND  Rinvest.buy < '10/01/97'
GROUP BY Rinvest.name

/* This view retrieves the name, average volume,*/
/* number of stocks, and the net value of the   */
/* investments in oil stocks of each investor   */

CREATE VIEW Oil(name, id, vol, cnt, value)
AS SELECT Rinvest.name, Rinvest.id,
          AVG(Wstock.vol), COUNT(Wstock.vol),
          SUM(Rinvest.qty*Wstock.price)
  FROM   Rinvest, Wstock
  WHERE  Rinvest.ticker=Wstock.ticker
    AND  Wstock.cat='Oil'
    AND  Wstock.date =  '11/03/97'
    AND  Rinvest.buy < '10/01/97'
GROUP BY Rinvest.name

/* The final select that makes use of the views */

SELECT   Oil.name, Oil.value, Tech.value
FROM     Oil, Tech
WHERE    Oil.id = Tech.id
  AND   (Oil.vol+Tech.vol)/(Oil.cnt+ Tech.cnt) >
        (SELELCT avg(Wstock.vol)
         FROM    Wstock
         WHERE  (Wstock.cat='Tech'
            OR  Wstock.cat='Oil')
           AND  Wstock.date = '11/03/97')
```

□

Figure 2 shows the query plan that is generated by the optimizer. The query plan consists of operators that take one or more plans as input and produce results that are consumed by the operators above.

Before closing this section, we note that our modeling of the plans, operators, and the properties, closely resemble the way most commercial relational systems implement their query engines [HFLP89].

4.2 Equivalence of Subplans

As mentioned in Section 1, our approach to optimization is based on analyzing the query plan generated by the optimizer for 'equivalent' subparts in the plan, and combining them into a transient view so that the redundancies in computation is avoided. However, our notion of equivalence is broad in that, the result produced by the combined plan is a superset of the results produced by the plans that are combined. We obtain the results corresponding to each of the combined plan by applying filters on the transient view. Below, we formalize some of these notions.

Definition 1 *Let D be a database, P be a plan, and $P(D)$ be the result obtained by applying P on D. We say plans P_1*

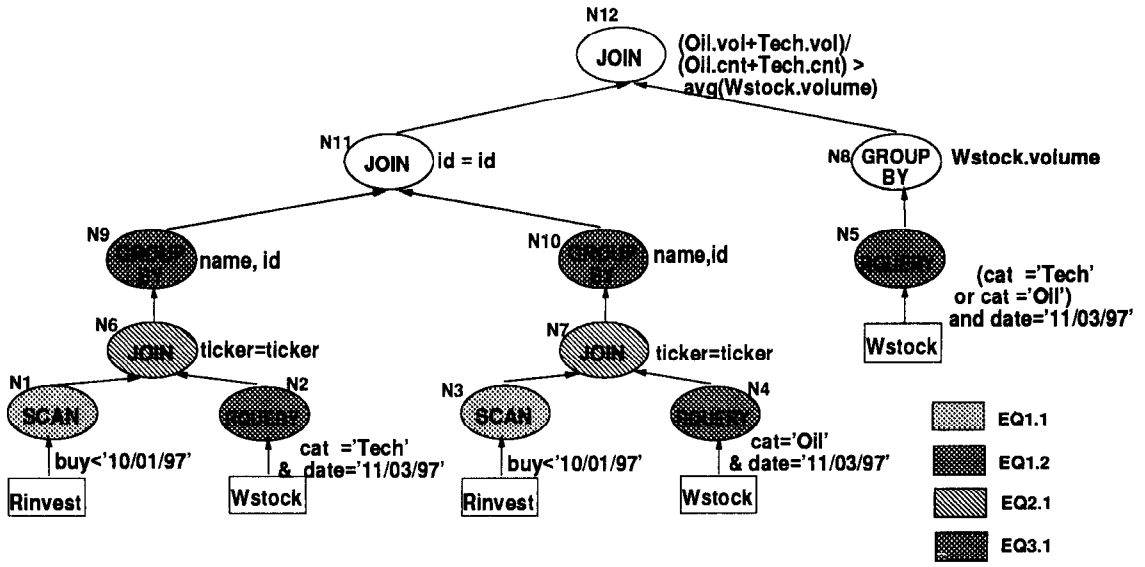

Figure 2: Query Plan

and P_2 are equivalent *if there exists a plan P, and expressions $E1$ and $E2$ that represent a sequence of application of the operators of Section 4.1 such that, for any database D, $P_1(D)$ and $P_2(D)$ can be obtained from $P(D)$ by applying $E1$ and $E2$ respectively. P is called a* covering plan *of P_1 and P_2.*

Intuitively, the above definition of equivalence is based on the notion of information preservation – the covering plan preserves the information corresponding to the plans it covers. The plans that are covered are considered equivalent.

To illustrate, two scan operators that operate on identical tables but apply different predicates would be considered equivalent in our approach. The plan that covers the scan operators is a new scan operator with an associated predicate that is an OR of the two predicates in the covered plans. The result associated with this plan is a superset of the result associated with each input scan operator. The results corresponding to the covered plans, can be obtained from that of the covering plan by applying the predicates associated with the covered scan operators as filter predicates.

Note that the problem of identifying equivalent subplans in a plan is essentially the problem of identifying equivalent subtrees in a tree. Naturally, we define equivalence based on induction, equivalence of nodes being the base case.

Definition 2 *Let n_1 and n_2 be two nodes of a plan tree T. We define a relation $\sim$ on nodes of T as follows. $n_1 \sim n_2$ if,*

case n_1 and n_2 are:

1. *Tables – n_1 and n_2 should be identical;*
2. *Plan Operators – their properties should agree as defined in Table 3, Table 4, Table 5, and Table 6*

The relation $\equiv$ is the symmetric, reflexive, transitive closure of $\sim$. Thus, $\equiv$ is an equivalence relation.

Note that by the above definition and the requirements imposed on the properties, the equivalence of two nodes n_1 and n_2 implies equivalence of the two subplans that respectively have n_1 and n_2 as their root.

Property	Condition
Scan type	Identical (eg:table or index)
Table scanned	Identical
Set of simple predicates	does not matter
Columns selected	does not matter

Table 3: Scan Properties

We now present the rationale behind our choice of the equivalence conditions.

- **SCAN :** A scan operator operates on one table and applies a set of predicates to the table and produces a set of columns as the result. Two scan operators are equivalent if they differ only in the columns they select and the simple predicates they apply[2]. The covering plan for the two scan operators is a new scan operator whose associated select list is a union of the columns selected by the covered operator and the columns in the predicate list of the two plans. The latter step is needed to ensure that we can apply the filter predicates. The predicate associated with the new scan operator is an OR of the covered operators' predicates. The filter query, to be applied on the covered plan, is obtained from the predicates of each of the covered scan operators (for selecting the appropriate subset) as well as the select list associated with the covered operators (for projecting the appropriate columns).
- **JOIN :** A join operator takes two plans as input and joins the two plans based on one or more predicates. We define two join operators to be equivalent if (i) the join method employed by them is identical, (ii) their inner plan belong to the same equivalence class, (iii) their outer plan belongs to the same equivalence

[2] Although, we require the scan type to be identical, we can relax this restriction for certain scan types. Refer to [SV98] for further details

Property	Condition
Join method	Identical
Outer plan	Equivalent
Inner plan	Equivalent
Join predicates	Identical

Table 4: Join Properties

Property	Condition
Input plan	Equivalent
Select list	does not matter
Aggregation Functions	does not matter
Group by list	subset and add columns from the simple predicate

Table 5: Group By Properties

class[3], and (iv) the join predicates are identical. Thus, the covering plan for the plan corresponding to the two join operators is a new join plan obtained from either of the covered plans by OR-ing the simple predicates in both the plans. The filtering query essentially consists of re-applying the simple predicates and projecting out the extra columns in the covering plan.

- **GROUP BY:** The group by operator applies aggregation functions to the columns of the input plan. Two group by operators are considered to be equivalent if (i) the input plans to the group by operator are in the same equivalence class and (ii) the columns in the group by list of one of the operators is a subset of the columns in the group by list of the other operator. The second restriction is to ensure that combining these two operators would indeed lead to a better performance. The covering plan is a new group by plan obtained from either of the covered plans by OR-ing the simple predicates in the covered plans and adding the columns mentioned in the simple predicates to the select as well as the group by list of the covering plan. The latter step is necessary to ensure that the covering plan computes the right results for the aggregation columns. The filtering query performs the right selections and projections (analogous to the case for the other operators) by deriving the necessary information from the predicates associated with the covered plans.

Property	Condition
Input plan	Equivalent
Data source where plan executes	Identical

Table 6: Rquery Properties

- **RQUERY:** The remote query operator represents the boundary with the remote database. It represents the point at which the results are fetched from the remote database. Two operators are equivalent if the input plans to the remote query operation are in the same equivalence class and the two queries operate on the same data source.

[3]For join methods such as merge join and hash join we loosen this restriction, the inner and the outer can be interchanged. Refer to [SV98] for further details

4.3 Equivalence Algorithm

In this section, we describe an efficient algorithm that given a query plan, identifies equivalent subplans in it. The algorithm makes use of the notion of equivalence ($\sim$) introduced in the previous section, and is based on a level-by-level walk of the query plan tree.

Algorithm 4.1 *Equivalence Class Generation*
Input: Query Plan Tree T
Output: Equivalence class of nodes in T with respect to $\sim$ (Definition 2)
Description: The algorithm is based on a level-by-level walk of the plan tree, starting with the leaf level, and identifying the nodes that are equivalent at that level, and using this information to identify equivalent nodes at the next level and so on.

```
begin
    cur_node = left most leaf;
    lvl = 0;
    While (cur_node != root) {
        while (there are more nodes at level lvl) {
            for (each unmarked node n at level lvl){
                if (cur_node ~ n) {
                    add_node_to_eq_class (lvl, cur_node, n)
                    // adds node n to equiv_class[lvl][cur_node]
                    mark (cur_node); mark(n);
                }
            }
            cur_node = next unmarked node at level lvl;
        }
        lvl = lvl +1; cur_node = leftmostnode at level lvl;
    }
end
```

Figure 2 illustrates a query plan along with the equivalence classes identified by the algorithm. The nodes representing the same equivalence class are similarly shaded.

5 Cost Considerations for Materializing Transient-Views

Algorithm 4.1 of Section 4, generates several equivalence classes – each class containing the equivalent nodes in a plan tree. A naive approach to implementing our idea would be to materialize one transient-view for each equivalence class, thus eliminating some of the redundancies present in the original plan. However, as we illustrate in Section 5.4, such an approach has the potential of degrading performance. In this section, we discuss how we leverage the existing costing facilities of a relational engine to estimate the cost of adopting a transient-views approach to optimization. We also present an algorithm that makes use of the equivalence classes to generate a set of transient-views such that it is guaranteed that there is no degradation in performance.

5.1 Transient-View Plan Generation

Our approach to estimating the cost of materializing the transient view involves two major steps: (1) Make use of the property information associated with each subplan in a equivalence class to generate an (SQL) query that defines the transient view corresponding to the equivalence class and (2) run the query through the optimizer to generate the (optimized) covering plan as well as the cost for materializing the transient-view. Below, we explain both these steps in detail.

We derive the query that defines a transient view corresponding to an equivalence class in the following way.

Property	Notation	View Property
Tables	T_{view}	T_i
Columns	C_{view}	$C_1 \cup C_2 \cup C_{pred1} \cup C_{pred2}$
Join predicates	JP_{view}	JP_i
Simple Predicates	SP_{view}	SP_1 OR SP_2
Aggregation	AF_{view}	$AF_1(Exp) \cup AF_2(Exp)$
Group by list	GC_{view}	$GC_1 \cup GC_2 \cup Addnl_i$
Data sources	S_{view}	S_i
Cost	$Cost_{view}$	To be determined
Cardinality	$Card_{view}$	To be determined

Table 7: Covering Plan Properties

First, we derive the property list associated with the covering plan. We make use of these properties to generate the query that defines the transient-view. We obtain the properties of the covering plan from the properties of the subplan in the equivalence class corresponding to the covering plan. If P_1 and P_2 are two plans in the equivalence class, Table 7 shows the properties of the covering plan P, derived from the properties of P_1 and P_2. We now present the rationale behind how we derive the covering plan properties from those of the subplans.

- **Tables:** All subplans in an equivalence class have identical tables represented by T_i. Hence, the covering plan has exactly the same set of tables.
- **Columns:** The columns in the select list of the covering plan is the union of the columns in the select list and the columns in the predicate list of the two subplans. We include the columns in the predicate list for the reasons that they play a crucial role in generating the filter predicates and for correctly computing the aggregation operation(s) present in the subplans.
- **Join Predicates:** The join predicates in the subplans are identical by definition. Hence this property carries over to the the covering plan, unchanged.
- **Predicates:** SP_i is the representation of conjuncts of simple predicates of the form *Col* RelOp *value* in the two plans. The predicate property of the combined plan, SP_{view} is obtained by ORing SP_1 and SP_2. If either SP_1 or SP_2 is null, then the predicate property of the combined plan is also null. We can then optimize the combined predicate by applying standard predicate optimization techniques, to remove redundancies, and merge the predicates. The column that are part of the predicates are also added to the select list and the group by list of the view.
- **Aggregation Function:** The aggregation function property of the covering plan is the union of the aggregation functions in the subplans.
- **Group By List:** The columns in the group by list, by definition, is a superset of the columns in the select list. The group by list of the covering plan is obtained as a union of the group by list in the subplans, and the columns that are part of any predicates mentioned in the subplans. The latter columns are included in order to correctly generate the filter predicates.
- **Data Source:** The data source for the plans in the equivalence class are identical, and carry over to the covering plan as well.

For the sake of clarity, our discussions above were based on obtaining the covering plan properties from the properties of two subplans. Note that the properties of a plan that covers an equivalence class consisting of more than two elements can be obtained by a successive application of the above technique.

The next step in our goal of obtaining the cost and cardinality information associated with materializing the transient-view, involves defining a SQL query based on the derived covering plan properties. Note that the covering plan properties contain enough information to obtain the underlying the SQL statement in a straight-forward manner. Below, we present the generic SQL query that is derived from the covering plan properties of Table 7.

SELECT $Cols_{view}$, $AF_{view}(Exp)$
FROM T_{view}
WHERE JP_{view}
AND SP_{view}
GROUP BY GC_{view}

The cost and cardinality information for materialing the transient-view is obtained by passing the above query through the optimizer module of the query processing engine. An important point to note is that, we optimize only the portion of the query representing the transient-view and not the entire query. The cost information thus obtained, is used to decide if materializing the transient-views would indeed lead to a better performance.

5.2 Filter Plans

The transient-view materialization cost, discussed in the previous section is only part of the overall cost associated with using our approach for optimization. The other part concerns the cost for filtering the transient-view for the results of the individual subplans. To obtain the filtering cost, we adopt a similar approach as before – that of defining queries corresponding to each of the filtering operation and running them through the optimizer.

The filter query for each subplan is defined based on the properties of the transient-view and the subplan. Below, we present a generic filter query for a subplan P_i, that generates the results corresponding to P_i from the transient-view.

SELECT $Cols_i$, $AF_i(Exp)$
FROM TRANSIENT-VIEW
WHERE SP_i
GROUP BY GC_i

We use the query optimizer to generate plans for the filter query. We use the plan information to generate executable code for the filters and the cost of the filter plan to determine if it is efficient to use transient-views. The cost property of the filter operation is represented by $FilterCost_i$.

5.3 Cost and Cardinality Information

We compare the cost of executing the subplans in the equivalence class to the cost of materializing the transient-view and applying filters. If the latter cost is less than the cost of executing all the subplans, the transient-view is materialized. The following equation shows the cost formula used to study the trade-off in materializing the transient views, when there are n subplans in an equivalence class.

$$COST_{transient-view} + \sum_{i=1}^{n} FilterCost_i < \sum_{i=1}^{n} Cost_i$$

5.4 Equivalence Class Pruning

The first phase of our algorithm (Algorithm 4.1) generates several equivalence classes at each level of the plan tree. However, as Example 5.1 below illustrates, plans in an equivalence class can potentially 'subsume' the plans in an equivalence class at a lower level. Thus, it is not meaningful to consider all the equivalence classes generated in the first phase of our algorithm for transient-view materialization. In this section, we discuss a strategy that helps prune the set of equivalence classes, for possible materialization as transient-views.

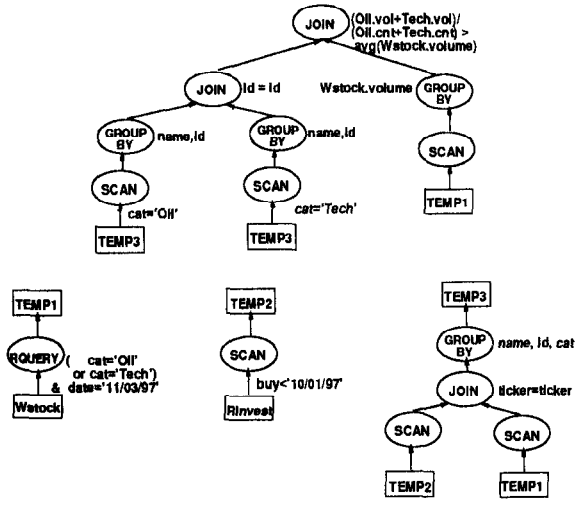

Figure 3: Liberal

Example 5.1 Figure 2 shows the plan generated by the query optimizer, and the set of equivalence classes generated by the first phase of the algorithm. Nodes belonging to the same equivalence class are shaded similarly.
Level 1: **EQ1.1:** {N1, N3}, **EQ1.2:** {N2, N4, N5} **EQ1.3:**N5
Level 2: **EQ2.1** {N6, N7}, **EQ2.2**{N8}
Level 3: **EQ2.3** {N9, N10}

For this query there are three equivalence classes at level 1. The equivalence class EQ2.1 at level 2 contains two JOIN operators that operate on the members of the level 1 equivalence class. The third level EQ2.3 has two group by nodes. Figure 3 shows the plan that results when we materialize all the equivalence classes. The transient-view materialized for EQ1.1 is wasteful, since the equivalence class EQ2.1 is a super-set of the nodes contained in EQ1.1.

□

Pruning Algorithm: We use the cost model and a hierarchical search algorithm to prune the classes for which we generate the transient-views. First, the cost of materializing the transient-view and the filter plans is computed for all the equivalence classes. If the cost of the transient-view and applying the filters is higher than that of the original plan, the equivalence class is pruned. To avoid materializing plans for equivalence classes that overlap, we use the following strategy.

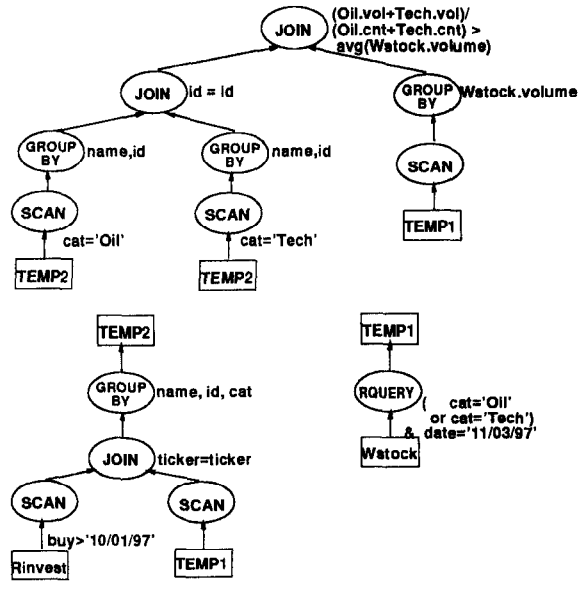

Figure 4: Plan Generated After Pruning Algorithm

We pick equivalence classes at the highest level (Level 3 in the above example), and add them to the *selected_list*. This ensures that we choose those equivalence classes that have nodes that share the most work in common. At subsequent levels, we prune equivalence classes for which all the nodes have at least one ancestor as members of the same equivalence classes in the *selected_list*. We add the remaining equivalence classes to the *selected_list*. We only materialize the views for the equivalence classes in the *selected_list*. This enables us to choose a non-overlapping set of equivalence classes. Figure 4 shows the plan that is generated when we use this algorithm to prune the equivalence classes. In this figure only the equivalence classes EQ2.1 and EQ1.2 are materialized. The algorithm for pruning the equivalence classes is described by Algorithm 5.1.

Algorithm 5.1 *Equivalence Class Pruning*
Input: Query Plan Tree T and equiv_class[][]
Output: Pruned list of equivalence classes for materialization
Description: The algorithm is based on computing the cost of all the equivalence classes and pruning those classes that do not improve performance. The algorithm also prunes those equivalence classes that overlap.

```
begin
For (each level lvl) {
    For (all equivalence class at each level i) {
        if (cost of materializing equiv_class[lvl][i] +
        filter plan cost ≥ sum of plan cost)
        prune equiv_class[lvl][i];
    }
}
lvl = highest level;
selected_list = null;
while (lvl ≥ 1) {
    for (all equivalence classes i in lvl) {
        common_ancst = selected_list;
        for (each plan p in equiv[lvl][i]) {
            equiv_ancst = equiv classes of ancestor nodes
            common_ancst = equiv_ancst ∩ common_ancst
            if (common_ancst is null) {
                add equiv[lvl][i] to selected_list;
```

```
                break;
            }
        }
        if (common_ancst not null) prune equiv[lvl][i];
    }
  lvl = lvl - 1;
  }
end
```

6 Performance

We implemented transient-views in DataJoiner [VZ97], a commercial heterogeneous database system, based on DB2. We used the TPCD benchmark queries to evaluate the performance impact of transient-views. Additionally, we formulated two queries based on the TPCD benchmark schema. We evaluated the queries on one processor RS6000 machines, on the AIX 3.2 platform. We ran the queries when the machines were idle (both the local and remote database) and report the average execution times of 10 runs.

The following is the outline of the rest of this section. First we briefly describe the TPCD schema in Section 6.1. Following this we describe the queries used to evaluate performance in Section 6.2. The performance results are discussed in Section 6.3, and the results are summarized in Section 6.5.

6.1 TPCD Benchmark Schema

The schema for the TPCD benchmark consists of a SUPPLIER(1000)the number in brackets indicates the cardinality of the table) table and PART(20000) table for parts supplied by the suppliers. The information of the SUPPLIER-PART relationship was maintained in the table PARTSUPP (80000)a. Information regarding customers was in a CUSTOMER (15000) table. Status information for orders placed by customers was maintained in the ORDER (150000) table. The key in the ORDER table was used to identify the parts and the suppliers from whom the part was ordered. This information was in a LINEITEM (600000) table, which also maintains shipping information, price, quantity ordered, and discount. The ORDER table also maintains the total price of the order, the order and ship priorities. We also had small NATION(25) and REGION(5) tables to associate suppliers and customers with the area they come from. For further information on the benchmark refer to the TPCD benchmark specification [TPC93]. The overall database size was around 100MB.

We split the TPCD tables and placed the SUPPLIER, NATION, and REGION table on a remote database accessed over the network. We placed the rest of the tables in the local DataJoiner database. DataJoiner accesses the remote tables through an user defined alias, eg: for SUPPLIER the alias was REMDB_SUPPLIER as though these were local tables. Even with placing the smaller tables at the remote database, we hope to show significant benefits for using transient-views. If we had placed the larger tales such as PART, PARTSUPP at the remote database, the benefits of transient-views would have been significantly better.

6.2 Queries to Evaluate Performance

Three queries, query 2, query 11, and query 15 were chosen from the TPCD benchmark suite. These three queries were chosen because, casual observation of these three queries suggest that these are likely to have redundant computation, and employing transient-views could be beneficial. We also evaluated the performance of the rest of the TPCD benchmark queries (query 1 through query 17) with the transient-view algorithm. The transient-view generation algorithm did not detect any redundancy in computation. Therefore, the plans of these queries remained unchanged. We also formulated two queries based on the TPCD schema to further illustrate the power of transient-views. We briefly describe the queries here.

- **TPCD Q02:** Find in a given region, for each part of a certain type and size, the supplier who can supply it at minimum cost. If several suppliers in that region offer the desired part type and size at the same (minimum) cost, the query lists the parts from the suppliers with the 100 highest account balances. For each supplier, the query lists the supplier's account balance, name, nation, the parts number and manufacturer, the supplier's address, phone number and comment information.
- **TPCD Q11:** This query finds by scanning the available stock of suppliers in a given nation, all the parts that represent a significant percentage of the total value of all available parts. The query displays the part number and the value of those parts in descending order of value.
- **TPCD Q15:** Find all the suppliers who contributed the most to the overall revenue for parts shipped during a given quarter of a year.
- **Query1:** Find the parts, whose size is 45 that represents a significant percentage of the value of the parts that are of size 48.
- **Query2:** Print the names of those parts that were given a 10% discount and also taxed 10% that were returned and the returns were not accepted.

6.3 Optimized Plan vs Transient-Views

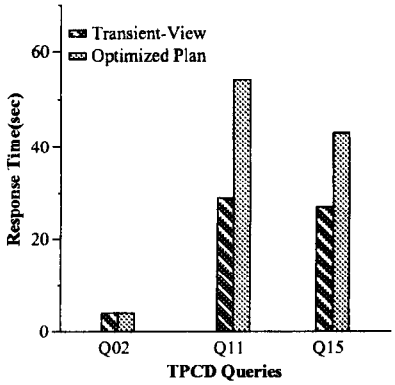

Figure 5: TPCD Query Performance

We evaluate the performance of transient-view on the three TPCD queries. Figure 5 shows the response times of the queries. The graph shows that using transient-views reduces the execution time for queries Q11 and Q15 nearly in half. Although the query syntax for Q02 suggests that it has redundancies, the transient-view mechanism does not detect any redundancies in the optimized plan. The reason for not detecting any redundancies will be better explained in in the next section.

Figure 6 shows the response time of the two queries that we formulated. Similar to the TPCD queries, the graph shows a 60% reduction in response time when transient-views were used for these two queries.

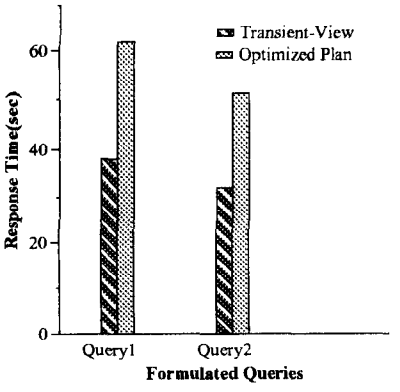

Figure 6: Query Performance

6.4 Transient-View vs Query Rewrite

In this section we demonstrate the detrimental effect of generating common subexpressions to eliminate redundancies by examining the query syntax [Hal76]. We compare the performance of TPCD-Q02 with transient-views and with factoring redundancies based on the query syntax.

For Q02, the common subexpression factoring algorithm based on query syntax identifies the join between PARTSUPP, REGION, SUPPLIER, and NATION as a common subexpression. By designating this as a common subexpression, the factoring algorithm reduces the join enumeration choices for the optimizer. The optimizer cannot join the PART table with the PARTSUPP table and make use of the index on PARTSUPP table. The performance of query execution degrades significantly. The execution time was 10 times worse than that of the optimized plan without factoring common subexpressions. On the other hand, the transient-view generation algorithm, does not detect any redundancies since the plan tree does not show any redundancies. Therefore, the performance remains the same.

This experiment demonstrates that identifying common subexpression from the query syntax can severely degrade performance, while transient-views does not affect the query execution times when the optimized plan does not indicate redundancies.

6.5 Summary of Results

In summary, our code has no impact on performance for queries that do not benefit from transient-views. The overhead of the algorithm to detect transient-views is negligible. Detecting and utilizing transient-views help improve performance tremendously as demonstrated by the reduction in half of the execution time of the two TPCD benchmark queries. The reason for this reduction is due to transient-views that avoids re-computing the results of a large subplan. If the subplan results are used multiple times within the same query, the performance benefits would be greater. Transient-views are used only when the performance benefits of using it is very clear. Comparison of employing transient-view with that of factoring common subexpressions based on query syntax shows that a query rewrite

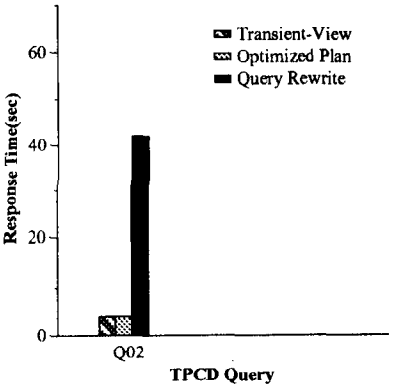

Figure 7: Query Rewrite vs Transient-View

mechanism limits the optimizers choices in choosing plans. This could lead to plans that severely degrade performance.

7 Conclusion

For many next generation database applications, the ability to integrate and reason over data from multiple, heterogeneous data sources is critical. Decision Support queries often involve processing huge amounts of data, stored in multiple, heterogeneous data sources, and often spread across geographical and other boundaries. Processing these queries involves repeated access of the same data sources, and sometimes multiple execution of similar processing sequences. In this paper we have described a cost-based optimization technique, based on a notion of transient-views, that enables us to remove redundant computation from the optimal plan generated by the optimizer. We define a broad notion of equivalence that enables us to capitalize on similarities in steps involved in processing a query. The novelty of our approach lies in the fact that multiple processing steps can be combined, executed as one step, and then compensated later – leading to an overall improvement in performance. Through performance evaluation experiments based on TPC-D benchmark queries, we show that the transient-view based approach leads to a significant improvement in performance of decision support queries. We have demonstrated the ease of implementing these algorithms by implementing it in a commercial heterogeneous database optimizer.

References

[Abi97] Abiteboul, Serge. Querying Semi-Structured Data. In *6th ICDT*, Delphi, Greece, Jan. 1997.

[AMMT96] Atzeni, Paolo, Mecca, Giansalvatore, Merialdo, Paolo, and Tabet, Elena. Structures in the web. Tech. report, DDS, Sezione Informatica, Universita di Roma Tre, 1996. Tech. Rep.

[ASD+91] Ahmed, R., Smedt, P., Du, W., Kent, W., Ketabchi, A., and Litwin, W. The Pegasus Heterogeneous Multidatabase System. In *IEEE Computer*, December 1991.

[CAS94] Christophides, V., Cluet, S. Abiteboul, S., and Scholl, M. From structured documents to novel query facilities. In *ACM SIGMOD*, 1994.

[CGH+94] Chawathe, S., Garcia-Molina, H., Hammer, H., Ireland, K., Papakonstantinou, Y., Ullman, J.D., and Widom, J. The TSIMMIS

Project: Integration of Heterogeneous Information Sources. In *IPSJ*, Tokyo, Japan, 1994.

[CKPS95] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing Queries with Materialized Views. In *ICDE*, March 1995.

[DFJ+96] S Dar, M. Franklin, B Jonsson, D. Srivastava, and M. Tan. Semantic Data Caching and Replacement. In *VLDB*, Mumbai, Sept 1996.

[Fin82] S. Finkelstein. Common Subexpression Analysis in Database Applications. In *ACM SIGMOD*, 1982.

[GBLP95] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. Tech. Rep. MSR-TR-95-22, Microsoft, 1995.

[GHRU97] H. Gupta, V. Harinarayanan, A. Rajaraman, and J. Ullman. Index Selection for OLAP. In *ICDE*, May 1997.

[GLS93] P. Gassner, G.M. Lohman, and Y Schiefer, B. Wang. Query Optimization in the IBM DB2 Family. *Data Engineering Bulletin*, 16(4), 1993.

[GM80] J. Grant and J. Minker. On optimizing the Evaluation of a Set of Expressions . Tech. Rep. TR-916, Univ. of Maryland, July 1980.

[GM81] J. Grant and J. Minker. Optimization in Deductive and Conventional Relational Database Systems. *Advances in Database Theory*, 1:195–234, 1981.

[Gup97] H. Gupta. Selection of Views to Materialize in a Data Warehouse. In *ICDT*, January 1997.

[Hal76] P.V. Hall. Optimization of a Single Relational Expression in a Relational Database System. *IBM Journal of R&D*, 20(3), May 1976.

[HFLP89] L. Haas, J. Freytag, G. Lohman, and H. Pirahesh. Extensible Query Processing In Starburst. In *ACM SIGMOD*, June 1989.

[HKWY97] L.M. Haas, D. Kossmann, E.L. Wimmers, and J. Yang. Optimizing Queries Across Diverse Data Sources. In *VLDB*, Aug 1997.

[HRU96] V. Harinaryanan, A. Rajaraman, and J.D. Ullman. Implementing Data Cubes Efficiently. In *ACM SIGMOD*, pages 205–216, May 1996.

[Jar84] M. Jarke. Common Subexpression Isolation in Multiple Query Optimization. *Query Processing in Database Systems, Springer Verlab*, pages 191–205, 1984.

[JV84] M. Jarke and Y. Vassiliou. Query Optimization in Database Systems. *ACM Computing Surveys*, June 1984.

[Kim84] W Kim. Global Optimization of Relational Queries. *Query Processing in Database Systems, Springer Verlag*, 1984.

[LMSS95] A.Y. Levy, A.O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering Queries Using Views. In *ACM SODS*, May 1995.

[LRO96] Levy, A.Y., Rajaraman, A., and Ordille, J.J. Querying Heterogeneous Information Sources using Source Descriptions. In *VLDB*, pg 251–262, 1996.

[LSS96] Lakshmanan L.V.S., Sadri F., and Subramanian, I. N. SchemaSQL – a language for querying and restructuring multidatabase systems. In *VLDB*, pg 239-250, Bombay, September 1996.

[PGW95] Papakonstantinou, Yannis, Garcia-Molina, H., and Widom, Jennifer. Object exchange across heterogeneous information sources. In *ICDE*, Taipei, Taiwan, February 1995.

[PHH92] H Pirahesh, J.M Hellerstein, and W. Hassan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *ACM SIGMOD*, pages 39–48, 1992.

[Rou82a] N. Roussopoulos. The Logical Access Path Schema of A Database. *IEEE Trans. on Soft. Engg.*, 8(6):563–573, Nov 1982.

[Rou82b] N. Roussopoulos. View Indexing in Relational Databases. *ACM TODS*, 7(2):258–290, June 1982.

[RSS96] K.A. Ross, D. Srivastava, and S. Sudarshan. Materialized View Maintainence and Integrity Constraint Checking: Trading Space for Time. In *ACM SIGMOD*,May 1996.

[SAB+95] Subramanian, V.S., Adali, S., Brink, A., Emery, R., Lu, J.J, Rajput, A., Rogers, T.J., Ross, R., and Ward, C. HERMES: Heterogeneous Reasoning and Mediator System. Tech. Rep., Inst. for Adv. Comp. Studies and Dept. of CS. Univ. of Maryland, 1995.

[SAC+79] P. Selinger, M. Astrhan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *ACM SIGMOD*, 1979.

[SDJL96] D. Srivastava, S. Dar, S. Jagadish, and A. Levy. Answering Queries with Aggregation Using Views. In *VLDB*, Sept 1996.

[Sel88] T.K Sellis. Multiple-Query Optimization. *ACM TODS*, 13(1):23–52, March 1988.

[SV98] N. Subramanian and S. Venkataraman. Cost based optimization of decision support queries using transient-views. Internal Report, 1998.

[TPC93] TPC. TPC Benchmark^{TM} D (Decision Support). Working draft 6.0, Transaction Processing Performance Council, August 1993.

[TRS97] M. Tork-Roth and P. Schwarz. Dont Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources . In *VLDB*, Aug 1997.

[TRV96] Tomasic, A., Raschid, L., and Valduriez, P. Scaling heterogeneous databases and the design of disco. In *IEEE DCS*, 1996.

[VZ97] S. Venkataraman and T. Zhang. DataJoiner Optimizer: Optimizing Query for Heterogenous Database Systems. In *Submitted for Publication*, 1997.

[WY76] E. Wong and K. Youssefi. Decomposition: A Strategy for Query Processing. *ACM TODS*, 1(3):223–241, Sept 1976.