Minimizing Data-Communication Costs by Decomposing Query Results in Client-Server Environments (Extended Version)

UCI ICS Technical Report, June 2003

Jia Li
Information and Computer Science
UC Irvine, CA 92697, USA
jiali@ics.uci.edu

Rada Chirkova
Department of Computer Science
North Carolina State University, Raleigh, NC 27695, USA
chirkova@csc.ncsu.edu

Chen Li
Information and Computer Science
UC Irvine, CA 92697, USA
chenli@ics.uci.edu

Abstract

Many database applications adopt a client-server architecture, in which data resides on a server that receives queries from a client. For each client's query, the server often needs to transfer to the client a large amount of data that is an answer to the query. The communication network in these environments could become a bottleneck in the computation. In this paper we study how to minimize the communication costs of transferring answers to large-join queries from server to client. We propose a novel technique that decomposes the answer into intermediate results, or views, which can reduce the redundancy in the answer. These views are transferred to the client and are used by the client to compute the final answer. There are several challenges in implementing this technique: (1) the number of possible plans to decompose the answers could be very large; (2) the technique requires an efficient algorithm to give an accurate estimate of the size of each view; and (3) many factors could affect the decomposition choice; one such factor is whether relevant data is cached on the client. Our extensive experiments on queries adapted from the TPC-H benchmark show that our technique can significantly reduce the communication costs of transferring answers to large-join queries. The extra steps used in our approach do pay off to reduce the total time of transferring the result of a query, when the result has a lot of redundancy.

1 Introduction

Many database applications use the client-server architecture, in which data resides on a server that receives queries from a client. For each query, the server sends the answer to the client. Applications that use this architecture include data integration, grid computing, and the recent "database-as-a-service" computing model [16]. Consider, for instance, mediators in data-integration systems [25]; mediators support seamless access to autonomous, heterogeneous information sources. Given a user query, a mediator translates the query to a sequence of queries on the sources, and then uses the answers from the sources to compute the final answer to the user query [14]. For each source query issued from the mediator ("client"), the results of the query are transferred from the source ("server") to the mediator. As another example, "database as a service" is a new model for enterprise computing, made possible by recent advances in networking and Internet technologies. In this model, companies and organizations choose storing their data on a server over having to maintain their local databases. The server provides its clients with the capabilities to create, store, modify, and query data on the server. When a client issues a query, the server uses the stored data to compute the answers and then sends the answers to the client over the network.

These applications adopt the client-server architecture and share the following characteristics. (1) Both the client and the server are able to do computation. The client might prefer computing some part of the query answer to receiving excessively large amounts of data from the server, as in the database-as-a-service scenario; alternatively, the client might have to do computation anyway, as in the mediation scenario. (2) The computation is data driven; the data resides on the server that is different from the client where a query is issued — either by client's choice, as in the database-as-a-service scenario, or by design, as in the mediation scenario. (3) The server needs to send data to the client over a network. When query results are large, the network could become a bottleneck; in addition, the client may want to minimize the costs of transferring the data over the network.

In the scenarios that have these three characteristics, minimizing data-transfer time becomes very important. In this paper we address the problem of minimizing communication costs incurred in transferring query results, in the applications that have these characteristics. We propose an efficient technique for reducing the communication costs: The idea is to decompose query answers into intermediate results, called "views," to reduce the redundancy in the data. The answers to these views are sent to the client; the client uses the answers to the views to compute the answer to the query. There are many possible ways (plans) to decompose query answers into view results; which plan is the best (i.e., has the minimum size) depends on the database on the server.

A critical advantage of our approach is that we use formal methods in [10] to choose provably globally optimal — and, at the same time, efficient — methods for decomposing queries into views. In other words, for a given query from client to server, our approach finds a view decomposition that provably minimizes the communication costs of transferring to the client the data for answering the query. Another feature of the proposed approach is that it can be implemented on top of the server database and thus make full use of the query-answering capabilities of the database-management system, including the capability to simultaneously process multiple queries from one or more clients. In particular, we can use the existing indexing structures in the server database to efficiently perform the required calculations. As shown in our experiments, in those scenarios where the communication cost dominates the total computation time, the extra time spent to evaluate the views does pay off in many cases, as our techniques significantly reduce the size of transferred data.

There are several challenges in implementing the proposed technique. (1) The number of possible plans to decompose query results into views could be exponential in terms of the number of relations in the query. (2) The technique requires an efficient approach to obtaining accurate view-size estimates; at the same time, traditional estimation techniques tend to give unreliable estimates for views defined as large-join queries. (3) Many factors could affect the process of finding a good plan, such as whether it is possible to use, in the computation of query answers, relevant data that is cached on the client.

In this paper we develop efficient solutions to these problems. The following are our contributions.

- We develop a novel approach to minimizing the communication costs of transferring query answers from server to client. For a given query from client to server, our approach finds a view decomposition that provably minimizes the size of data transferred to the client for answering the query. Thus the total time to send the result is reduced when the transfer cost is high (Section 3).
- We propose rules for reducing the search space of optimal decomposition plans. The rules can take into account the cached data on the client, which allows us to further reduce the communication costs of sending the results to the client (Section 4).
- We study how to efficiently and accurately estimate the sizes of views (Section 5).
- Our extensive experiments on queries adapted from the TPC-H benchmark show that our technique can efficiently and significantly reduce communication costs (Section 6).

1.1 Related Work

The original motivation for finding views to materialize to answer queries comes up in the context of designing data warehouses. The problem of finding the "right" views to materialize has traditionally been studied under the name of view selection (e.g., [1]). Recently, [8, 9] proposed an approach to finding views to materialize that considers all possible views that can be invented to optimize a given metric of database performance. That approach emphasizes formal methods in finding a set of views that are provably globally optimal for a given query or queries and for a given performance metric. In this paper, we extend the approach to the metric of minimizing the communication costs in client-server scenarios.

Minimizing data-communication costs is one of the key problems in distributed database systems [5, 19]. Heuristic search techniques have been proposed for finding a sequence of joins and semijoins that reduce the communication costs in distributed query processing [2, 4, 6]. In a semijoin, it is assumed that relations to be joined are at different sites. All local operations such as selections, projection and joins are performed first, then a projection on join attributes is shipped between two sites to remove the dangling tuples. In our case, relations to be joined are all stored at the server site. Our approach is to project the query results on the server site to different views, so that global dangling tuples are removed among these views. Besides, because everything is on the server, we can do more than the local selections, that is, both local and global selections among the views, to get more reduction of view sizes. One challenge in our approach is how to find the optimal way to do the projections.

[18] discusses the efficiency of distributed joins in System R^*. [22] describes the Mariposa system to deal with the mobility of data stored in different sources. [14] presents the design of a cost-based optimizer for heterogeneous middleware systems where data sources may have different query capabilities. [20] addresses how to determine the costs of query plans in a wrapper-based heterogeneous system. The approach uses the information provided by query-plan nodes to eliminate some costly plans early. The cost model in the approach includes communication costs, which are modeled via stored constants. Yet our approach is more flexible, as it takes into account features of specific queries.

These previous *distributed* approaches focused on the problem of query placement to *different* sources. That is, they decompose a query into subqueries among individual server nodes. Once the query-execution location is fixed and subqueries reach individual servers, our approach presented in this work can be applied to do the server-level (sub)query optimization. In other words, our approach is applicable on top of the traditional local query optimizer, which further reduces the communication costs between the sources nodes and the query site. In addition, by considering views to construct the query results, our approach could utilize possible cached data on the client with more flexibility.

Another related topic is the hybrid shipping model in distributed query processing [11]. That model allows a server to execute portions of an access plan on a client, but if the server is asked to return two views that are to be joined in the client, they do not pre-compute the join of the two relations, so that all useless tuples that do not join with other views, i.e., dangling tuples, are still kept and sent to the client, causing extra network load. Our approach also leaves some computation to clients. But on the server site, we further reduce the size of data transferred by utilizing all potential joins to be performed on clients, which is similar to what a semijoin does among different server sites. So in our approach, the views to be transferred no longer contain those dangling tuples, thus leading to further reduction of communication cost. In addition, past work in distributed query processing does not emphasize the study of global optimality guarantees of the proposed solutions. In contrast, our approach can choose provably globally optimal plans to minimize the communication costs.

[17] describes methods for converting relational data into nested structures; one of the methods decomposes relational query results into relational views. Although they do make use of all join information to reduce the redundancy and eliminate useless data in the result, their view decomposition, for each input, always considers (and produces) just one *fixed* set of views; that is, one for each relation. In addition, their approach is inefficient in getting the result data by using a nested-loop join. Finally, their approach does not have optimality guarantees for the outputs produced by the algorithm.

Data-compression techniques (e.g., [23, 7]) can be used to compress the results for efficient transfer. As we will see in Section 6.6, our approach is orthogonal to those compression techniques. In particular, those techniques can be applied to the results of our approach, to further reduce the data-communication costs.

In [10] the general problem of finding an optimal view decomposition for a query result is studied. It is shown that when the query has self-joins, we need to consider disjunctive views as an optimal solution. In this paper, we focus on the case where an optimal solution includes conjunctive views only. We study how to find the views efficiently, and experimentally evaluate our techniques.

2 Problem Formulation

In this section we formulate the problem of minimizing the data-communication costs in transferring the results of large-join queries, by decomposing the query answers into intermediate view results. We first present and discuss a motivating example and then give a formal specification of the problem.

2.1 A Motivating Example

Consider the following simplified versions of three relation schemas in the TPC-H benchmark [24]:

```
customer(custkey(4),name(25),mktsgmt(10))
order(orderkey(4),custkey(4),orderdate(8),
      shippriority(4),comment(79))
lineitem(linenumber(4),orderkey(4),
         quantity(4), shipmode(10))
```

The underlined attribute of each relation is a primary key for this relation. The number after each attribute is the size of the values of the attribute, in bytes. (For simplicity, we assume for each attribute that all its values are of the same size.) There is a referential-integrity constraint from attribute `order.custkey` to attribute `customer.custkey`. We further assume that the relations reside on a server that accepts queries from a client.

Suppose a user at the client issues a query Q_1, shown in Figure 1. Q_1 is a variation on the Query 3 in the TCP-H benchmark. The server computes the answer to Q_1 and

```
SELECT   c.name, o.orderdate,
         o.shippriority, o.comment,
         l.orderkey, l.quantity, l.shipmode
FROM     customer c, orders o, lineitem l
WHERE    c.mktsgmnt = 'BUILDING'
         AND c.custkey = o.custkey
         AND o.orderkey = l.orderkey
         AND o.orderdate < '1995-03-20'
         AND l.shipdate > '1995-03-20';
```

Figure 1. Query Q_1.

sends it back to the client. Suppose there are 4000 tuples in the answer to Q_1 on the database at the server.[1] It follows that the total number of bytes sent to the client is:

$$4000 \times (25 + 8 + 4 + 4 + 79 + 4 + 10) = 516,000.$$

Let us see if we can reduce the communication costs by reducing the amount of data to be transferred to the client, while still giving the client all the data it needs to compute the final answer to the query Q_1. Table 1 shows a fragment of the answer to Q_1; it is easy to see that the answer to Q_1 has redundancies. For instance, tuples t_1 through t_3 are the same except in the values of `l.quantity` and `l.shipmode`; tuples t_4 and t_5 have similar redundancy. One reason for the redundancy is that an order could have several lineitems with different quantities and shipmodes. In the join results, these orders data generate several tuples with the same values of `customer` and `order`. Based on this observation, we can decompose the answer to Q_1 into intermediate results — *views* V_1 and V_2 — as shown in Figure 2. We will obtain the answer to the query Q_1 by joining the views.

Assume there are 1200 tuples in view V_1 and 4000 tuples in view V_2. By using the sizes of the attribute values in the answers to the two views, we obtain that the total size of the answers to the views is $216,000$ bytes. Recall that the size of the answer to the query Q_1 is 516,000 bytes, and that it is possible to compute the answer to Q_1 using the answers to the views V_1 and V_2. It follows that instead of sending the client the (large) answer to the query Q_1, the server can reduce the transmission costs by sending the client the results of the two views; the client can then use the view results to compute the answer to the query.

This example shows that it is possible to decompose queries into intermediate views, such that the answers to the views can be used to compute the exact answer to the query, and the total size of the answers to the views can be much smaller than the size of the answer to the query. At the same time, we make the following observations. (1) When

```
  View V1:
SELECT   c.name, o.orderdate, o.shippriority,
         o.comment, o.orderkey
FROM     customer c, orders o, lineitem l
WHERE    c.mktsgmnt = 'BUILDING'
         AND c.custkey = o.custkey
         AND o.orderkey = l.orderkey
         AND o.orderdate < '1995-03-20'
         AND l.shipdate > '1995-03-20';
  View V2:
SELECT   l.orderkey, l.quantity, l.shipmode
FROM     customer c, orders o, lineitem l
WHERE    c.mktsgmnt = 'BUILDING'
         AND c.custkey = o.custkey
         AND o.orderkey = l.orderkey
         AND o.orderdate < '1995-03-20'
         AND l.shipdate > '1995-03-20';
```

Figure 2. Decomposing the query answer to two views.

trying to reduce the redundancy in the query answers, we may need to add more attributes that will allow joins of the view results. (2) There is more than one way to decompose the answer into views. (3) If the client has, in the cache, the answers to previously asked queries, then the cached data can be used to further reduce the communication costs.

2.2 Problem Specification

Consider a client-server environment that uses a relational database. All base relations are stored on the server, which takes queries from the client. Given a query submitted to the server, we try to reduce the communication costs of transferring the query answer to the client, by decomposing the answer into intermediate results, or *views*. Intuitively, the answers to the views in a good decomposition have less redundancy than the query answer. The server sends the view answers to the client, and the client uses the view answers to compute the final answer to the query. In this paper, we focus on select-project-join (SPJ) queries and views in the following format:

> `SELECT` $A_1, \ldots, A_n$
> `FROM` $R_1, \ldots, R_m$
> `WHERE` $cond_1$ `AND` $cond_2$ `AND` ...`AND` $cond_k$;

Attributes $A_1, \ldots, A_n$ are called the *output attributes* of the query/view. Each condition $cond_i$ is in the form A *op* X, where A is an attribute, X is a constant or an attribute, and *op* is one of the following operators: $<$, $\leq$, $=$, $\geq$, $>$, `LIKE`. The results we report in this paper use the set-semantics assumption for computing query answers. We consider two types of conditions

[1] In this paper we assume set semantics, i.e., no duplicates exist in query answers.

Table 1. Partial results of query Q_1.

Tuple id	c.name	l.orderkey	o.comment	o.shippriority	o.orderdate	l.quantity	l.shipmode
t_1	Tom	134721	closely ironic ... [79 characters]	0	3/14/1995	26	REG AIR
t_2	Tom	134721	closely ironic ... [79 characters]	0	3/14/1995	75	REG AIR
t_3	Tom	134721	closely ironic ... [79 characters]	0	3/14/1995	43	AIR
t_4	Jack	571683	final sentiments ... [79 characters]	0	12/21/1994	43	MAIL
t_5	Jack	571683	final sentiments ... [79 characters]	0	12/21/1994	33	AIR
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

$A\ op\ X$. (1) A *join condition* $A\ op\ X$ uses *join attributes* — two attributes from two different relations. Examples of join conditions are: `c.custkey = o.custkey`; `o.orderkey = l.orderkey`. (2) All other conditions $A\ op\ X$ are *selection conditions*, e.g., `o.orderdate < '1995-03-20'`.

Given a query Q on the database on the server, a *decomposition plan* $P = \{V_1, \ldots, V_k\}$ for the query is a set of intermediate views $V_1, \ldots, V_k$, such that the client can compute the answer to the query Q using the answers to the views. The *size of a decomposition plan* $P = \{V_1, \ldots, V_k\}$ is $size(P) = \sum_{V_i \in P} size(V_i)$, where each $size(V_i)$, the size of view V_i, is the total number of bytes in the answer to the view. For simplicity, we assume that all tuples in a view have the same size; this size is the sum of sizes, in bytes, of the corresponding attributes.

Problem Statement: Given a query on a database at the server, find an *optimal decomposition plan*, which is a decomposition plan with a minimum size.

Notice that an optimal plan for a query is computed on a fixed database instance and is therefore dependent on the database of the server. Our goal, given a database, is to find an optimal plan efficiently. After finding an optimal plan, the server sends the results of the views in the plan to the client. As the view results are streaming in, the client can compute the query answer in a *pipelined* manner. By using the pipelining operators we can reduce the total execution time for the client to get the final answer.

3 Choosing Views

In this section we first define the view space that needs to be considered to form plans for a query Q, and discuss one method for computing these views efficiently.

3.1 Determining the Search Space of Views

Given a query Q, there are many possible views that can be used to compute the answers to the query. Not all of these views need to be considered in the search for an optimal decomposition plan, since the sizes of some views are greater than the sizes of other views, no matter what the database is. In this section we define a class of views that we consider in decomposition plans of a query Q. We will see that this class includes all views that need to be considered to find an optimal plan for the query Q.

We begin by defining those attributes in the query Q that can be output attributes of views in some decomposition plan. Consider an arbitrary subset $G = \{S_{i1}, \ldots, S_{ik}\}$ of the relations in the `FROM` clause of the query Q; let G' be the remaining relations in Q. An attribute A of a relation in G is a *relevant* attribute of G if either (1) A is an output attribute of Q, or (2) A is a join attribute used in a join condition $A\ op\ X$ (in Q), where X is an attribute of a relation in G'. The *associated view of* G, denoted V_G, is defined as:

`SELECT` [all relevant attributes of G]
`FROM` [all relations in the `FROM` clause of Q]
`WHERE` [all conditions in the `WHERE` clause of Q];

The view V_G has the same `FROM` and `WHERE` clauses as the query Q. The only difference is that in the answer to V_G, all attributes are relevant attributes of G. Intuitively, V_G needs its join attributes for joins with other views in computing the query's answers, and the output attributes of V_G in the query's answer.

For example, the view V_1 in Figure 2 is associated with the subset G_1=\{`customer`, `order`\}. The relevant attributes of the view V_1 include output attributes of the query Q_1 — `c.name, o.orderdate, o.shippriority, o.comment` — as well as a join attribute `o.orderkey`. Table 2 shows all possible subsets of the relations in the query Q_1 that need to be considered in searching for an optimal decomposition. (We will show in Section 4 that some of these views do not need to be considered.)

Lemma 3.1 *By considering the decomposition plans that use the views in the format above, we will always find an optimal plan for Q in the space of conjunctive views.* □

3.2 Computing the Views Efficiently

In order to search for an optimal plan in the space of views described in Section 3.1, we need to know the size of

Table 2. Possible views for query Q_1.

Relation subset	Associated view
$\{customer\}$	W_1
$\{order\}$	W_2
$\{lineitem\}$	$W_3 = V_2$
$\{customer, order\}$	$W_4 = V_1$
$\{customer, lineitem\}$	W_5
$\{order, lineitem\}$	W_6
$\{customer, order, lineitem\}$	$W_7 = Q_1$

each view. One brute-force approach is to get the exact sizes of all these views, by computing their answers at the server. This approach is clearly computationally prohibitive. Another approach is to estimate the size of each view by using traditional estimates of the query optimizer and statistics about the relations. Unfortunately, this approach tends to give inaccurate results as the number of joins in the view definition increases.

Now we propose a technique that allows us to estimate, both efficiently and accurately, the sizes of all views. Intuitively, we are able to obtain all the estimates by computing the answer to just one query. We define this query by adding, to the `SELECT` clause of the query Q, all relevant attributes of all views in our search space of views. All of the view-size estimation afterwards is based on the results from this extended query.

Formally, before searching for an optimal plan for a query Q, we first rewrite the query as follows. To the `SELECT` clause of the query Q, we add all its join attributes that are not in its output attributes. The new query is denoted $\hat{Q}$. We execute this new query on the server to get its results, denoted $D_{\hat{Q}}$. For instance, for the query Q_1 in Figure 1, we add to this query the join attributes: `c_custkey`, `o_custkey`, and `o_orderkey`, because they are not output attributes of the query Q_1. The new query is shown in Figure 3.

```
SELECT   c.name, c.custkey, o.custkey,
         o.orderdate, o.shippriority,
         o.comment, o.orderkey, l.orderkey,
         l.quantity, l.shipmode
FROM     customer c, orders o, lineitem l
WHERE    c.mktsgmnt = 'BUILDING'
         AND c.custkey = o.custkey
         AND o.orderkey = l.orderkey
         AND o.orderdate < '1995-03-20'
         AND l.shipdate > '1995-03-20';
```

Figure 3. Extended query $\hat{Q}_1$ of Q_1 (with added attributes in bold face).

We execute the query $\hat{Q}$, rather than the original query Q, for cost estimation. Since we already have the answer $D_{\hat{Q}}$ to the extended query $\hat{Q}$, we can use $D_{\hat{Q}}$ to estimate the size of each view, by sampling $D_{\hat{Q}}$ instead of the whole database. The resulting estimates in this way are likely to be more accurate than those obtained using standard size-estimation functions and database statistics. In Section 5 we discuss how to estimate view sizes efficiently and accurately.

Because the query $\hat{Q}$ is obtained by adding attributes to the query Q, computing the answer to $\hat{Q}$ could be more expensive than computing the answer to Q. However, as our experiments will show, this overhead is minor, because the new query does not change the join conditions, the evaluation of which often dominates the cost. Furthermore, we will see shortly that the overhead can pay off in the search for an optimal plan.

4 Efficient Search for an Optimal Plan

In this section, we study how to find an optimal decomposition plan for a query Q using the views defined in Section 3. We consider all partitions of the relations in the query Q. Each partition $T = \{G_1, \ldots, G_m\}$, which is a set of nonoverlapping subsets of the query's relations, yields a decomposition plan $P_T = \{V_{G_1}, \ldots, V_{G_m}\}$, where each view V_{G_i} is the associated view of G_i (Section 3.1).

Lemma 4.1 *By considering the decomposition plans corresponding to all partitions of the relations in query Q, we will always find an optimal plan for Q in the space of conjunctive views.* [2] $\square$

There are still challenges in implementing this approach. (1) The number of possible decomposition plans could be exponential in terms of the number of relations in the query. Thus, doing an exhaustive search to find an optimal plan could be computationally prohibitive. (2) We need to obtain an accurate estimate of the size of each view V_{G_i}. We discuss an efficient and accurate view-size estimation method in the next section. In this section, we address the first challenge. We assume the server has computed the answer $D_{\hat{Q}}$ to the extended query $\hat{Q}$ of Q; we are using the answer $D_{\hat{Q}}$ in the algorithm below. We first give a basic version of the search algorithm, then propose three pruning rules to reduce the search space.

4.1 Basic Version of the Algorithm

Figure 4 shows the basic version of the search algorithm. In the first step, the algorithm estimates the size of each view, using the technique discussed in Section 5. All view

[2]The correctness is formally proved in [10].

sizes are stored in a table S. In the second step, the algorithm considers all partitions of the query relations. For each partition T, it computes the total size of the views for the subsets in T. The algorithm searches for a minimum-size partition and generates the corresponding decomposition plan.

```
Input:
    • $Q$: original query
    • $D_{\hat{Q}}$: answer to the extended query $\hat{Q}$ of $Q$.
Output: a decomposition plan of $Q$ with minimum size.
Method:
  Step 1: // estimate view sizes
          initialize view-size table $S$ to empty;
          for each subset $G$ of relations in $Q$ {
              estimate size $S_G$ of view $V_G$ using $D_{\hat{Q}}$;
              add $(V_G, S_G)$ to table $S$;
          }
  Step 2: // find an optimal decomposition plan
          minSize = $\infty$; optPlan = {};
          for each partition $T$ of the relations in $Q$ {
              size = 0; plan = {};
              for each relation subset $G \in T$ {
                  get size $S_G$ by looking up table $S$;
                  size = size + $S_G$;
                  plan = plan $\cup\{V_G\}$;
              }
              if (size $<$ minSize) {
                  minSize = size;
                  optPlan = plan;
              }
          }

  return (optPlan, minSize);
```

Figure 4. Basic version of the algorithm.

As the number of relations in the query increases, the number of partitions considered by the algorithm becomes very large. Searching all these plans exhaustively is very expensive. In the next three subsections we propose three pruning rules that reduce the size of the search space while still allowing us to find an optimal plan. To illustrate the use of these pruning rules, we use the query Q_2 shown in Figure 5. Q_2 is a slight variation on Query 2 in the TPC-H benchmark. The following are simplified schemas of the relations used in the query; the underlined attribute(s) of each relation form(s) a primary key for this relation.

```
part(partkey,mfgr,type,size)
partsupp(partkey,suppkey,availqty)
supplier(suppkey,name,nationkey)
nation(nationkey,name,regionkey)
region(regionkey,name)
```

Before presenting the rules, we define the concept of "join graphs" [12, 27] that will be used in the pruning rules. Formally, the *join graph* of a query Q is a directed graph $G(Q) = (V, E)$, in which V is the set of relations used in Q, which form the vertices in the graph. There is a directed edge e: $R_i \rightarrow R_j$ in E if the query has a join condition $R_i.B = R_j.A$, and A is a key of R_j. The edge is annotated with "RI" if there is a referential-integrity constraint from $R_i.B$ to $R_j.A$.

```
SELECT   p.mfgr, ps.partkey, ps.suppkey,
         s.name, n.name
FROM     part p, supplier s, partsupp ps
         nation n, region r
WHERE    p.partkey = ps.partkey
         AND s.suppkey = ps.suppkey
         AND p.size = 24
         AND p.type LIKE 'STANDARD %'
         AND s.nationkey = n.nationkey
         AND n.regionkey = r.regionkey
         AND r.name = 'AMERICA';
```

Figure 5. Query Q_2

As an example, the join graph of the query Q_2 is shown in Figure 6. The edge `nation` $\rightarrow$ `region` in the graph represents the fact that there is a referential-integrity constraint from `nation.regionkey` to `region.regionkey`, and that there is a join condition

```
nation.regionkey = region.regionkey
```

in query Q_2. Such an edge is called an "RI-join edge."

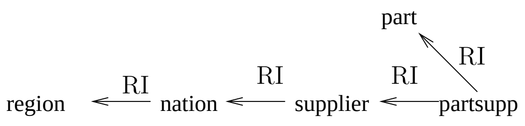


Figure 6. Join graph $G(Q_2)$ of query Q_2.

Before running our search algorithm, we first extend Q_2 by adding, to the `SELECT` clause, its join attributes that are not output attributes, such as `p.partkey`, `n.nationkey`, etc. We execute the extended query $\hat{Q}_2$ on the server, and get the results $D_{\hat{Q}_2}$. Because the `FROM` clause of the query Q_2 has five relations, the number of plans for Q_2 considered by the basic version of the algorithm is $\sum_{i=1}^{5} S(5, i) = 52$, where $S(m, n)$ is the Stirling number.

4.2 Pruning 1: Ignoring Plans that Use "Noninterleaving" Views

Using this pruning rule, we never consider one class of views, which we call "noninterleaving views." As discussed in Section 3.1, given a query, we associate one view with each subset G of the relations in the `FROM` clause of the query. *Noninterleaving views* correspond to those subsets G that can be partitioned into two groups of relations, $G^{(1)}$ and $G^{(2)}$, such that the query does not have join conditions between the relations in the two groups. Rather than considering a noninterleaving view for a subset G of relations, we can always project the answer to the extended query on the relevant attributes of the groups $G^{(1)}$ and $G^{(2)}$ *separately*. In addition, in any decomposition plan for the query, the two views for the groups $G^{(1)}$ and $G^{(2)}$, taken together, have the same functionality as the noninterleaving view for the subset G. After we apply this rule to reduce the number of plans to be considered, the search space of the remaining views can still be prohibitively large: A query with n relations will still have up to 2^{n-1} possible plans. For Q_2, we still have 2^4, that is, 16 possible plans. Luckily, we have more rules to do the pruning.

4.3 Pruning 2: Removing Ear Relations without Output Attributes

Let us look at the `region` relation in query Q_2. This relation does not have any output attributes in Q_2, and its only join condition `r.regionkey = n.regionkey` involves another relation, `nation`. The only contribution of `region` to the query is in providing a selection condition `r.name = 'AMERICA'` and a join condition `n.regionkey = r.regionkey`. Therefore, we can ignore the `region` relation when enumerating relation subsets for views. Therefore, we can further reduce the number of possible plans for Q_2 to only $2^3 = 8$.

In general, we look for *ear relations* that do not have output attributes in Q. A relation is an ear relation if all its join attributes are shared with just one other relation [13]. If an ear relation does not have any output attribute, then its only contribution to the query's results is to provide selection or join conditions in the query. After applying these conditions, we can ignore this relation when generating relation subsets for views. (Notice that the ignored relation's conditions still remain in Q; we just do not consider it when enumerating relation subsets.) By ignoring an ear relation R, we could make another relation R' an ear relation in a more general sense, i.e., R' does not have any output attributes of Q, and all its join attributes are shared with either R or another remaining relation. We repeat this process until we cannot eliminate any more ear relations that do not have output attributes. Notice if a relation does not have any output attributes, but is connected with more than one other relation, we may not be able to remove it, since it could be "connecting" two other relations.

4.4 Pruning 3: Using Referential-Integrity Constraints

This pruning rule is based on referential-integrity constraints in the query relations. As specifying referential-integrity constraints is common in database design, this pruning technique is widely applicable. We use our running example, query Q_2, to illustrate the intuition of the rule. After ignoring the `region` relation in enumerating relation subsets (using pruning rule 2), we noticed that for *any* database on the server, no matter how relations `nation`, `supplier`, and `partsupp` were grouped in views, the data-size difference between the following two plans was always a constant: (1) a plan with `partsupp` and `part` in the same relation subset, (2) a plan that uses the same partition of the relations, except that `partsupp` and `part` were in different subsets. In particular, consider the following decomposition plans.

Plan	Relation subsets of views in the plan
P_1	$\{nation, supplier\}, \{partsupp\}, \{part\}$
P_2	$\{nation, supplier\}, \{partsupp, part\}$
P_3	$\{nation\}, \{supplier\}, \{partsupp\}, \{part\}$
P_4	$\{nation\}, \{supplier\}, \{partsupp, part\}$
P_5	$\{nation\}, \{supplier, partsupp\}, \{part\}$
P_6	$\{nation\}, \{supplier, partsupp, part\}$
P_7	$\{nation\ supplier, partsupp\}, \{part\}$
P_8	$\{nation\ supplier, partsupp, part\}$

Interestingly, independently of the database,

$$\begin{aligned} & size(P_1) - size(P_2) = size(P_3) - size(P_4) \\ = \; & size(P_5) - size(P_6) = size(P_7) - size(P_8) \end{aligned}$$

is always true. Therefore, we can *locally* decide whether to consider any view whose relation subset includes both `partsupp` and `part`, by comparing the sizes of the views for $\{\{$`partsupp`$\}$, $\{$`part`$\}\}$ and for $\{\{$`partsupp`, `part`$\}\}$. If the latter is smaller, then among all plans we only need to consider those plans that include just one view that corresponds to these two relations. Otherwise, we only need to consider the other 4 plans.

This relationship is not a coincidence. By analyzing the query closely, we found the following reasons: (1) the `partsupp` relation has a directed RI-join-edge path to each relation in the join graph; (2) this relation has a key that is among the query's output attributes (namely, `partkey` and `suppkey`); and (3) the `part` relation is a "leaf" relation joined with the `partsupp` relation. That is, in the join graph, the relation `part` is connected to only one relation. (In general, a leaf relation is always an ear relation for a query, but an ear relation might not be a leaf relation.) Now we generalize this relationship as follows.

Lemma 4.2 *Consider two relations R_1 and R_2 in a query Q, if they satisfy the following conditions:*

- *R_1 has a directed RI-join-edge path to each relation in the join graph $G(Q)$.*
- *R_1 has a key that is among the output attributes of query Q.*
- *R_2 is a leaf relation joined with R_1.*

For a given database, consider any two plans P_1 and P_2 that are the same except for the following: R_1 and R_2 are in the same relation subset in P_1 and in different relation subsets in P_2. Then $size(P_1) - size(P_2)$ is a constant, independently of how other relations of Q are grouped in the plans. □

We call the relation R_1 in the lemma the *core relation* of the query (see [3] for the related notion of pivot relation), and the RI-join edge from relation R_1 to R_2 an *independent edge*. Not all queries have core relations. And if a query does have a core relation, such core relation might be not unique, say, the query might have more than one core relations. If a query has an independent RI-join edge from R_1 to R_2, as specified in Lemma 4.2, we can use the edge to prune plans in the search space. In particular, when searching for an optimal plan, we compare $size(\{R_1\}) + size(\{R_2\})$ to $size(\{R_1, R_2\})$. Once we determine which of them is smaller, say, $size(\{R_1\}) + size(\{R_2\})$, we do not need to consider views in which relations R_1 and R_2 are in the same relation subset.

In summary, the three pruning rules can help us dramatically reduce the size of the search space *without sacrificing the optimality* of the outcome. We will explore these advantages experimentally in Section 6.

5 Estimating View Sizes

In order to find an optimal decomposition plan, we need to estimate the size of each view V. The size of the view V can be computed as: $size(V) = |V| \times (size\ of\ each\ tuple\ in\ V)$, where $|V|$ is the number of *distinct* tuples in the view V. The size of each tuple is the sum of sizes of the attributes of the view. All that remains to be computed is the number of distinct tuples in the view V. (Recall that we are using set semantics in the query results.) This number is estimated by projecting the sampled tuples from $D_{\hat{Q}}$ onto the relevant attributes in the view, and applying an estimator. In this section we discuss how to estimate the number of distinct tuples in view relations, efficiently and accurately.

5.1 Finding Views with the Same Number of Distinct Tuples

We observe that certain views always have the same number of distinct tuples, independently of the database. For instance, consider the query Q_2 in Figure 5. We can show that for the view {`partsupp`} for the query, the number of distinct tuples in the view is the same as in any view whose relation subset contains this relation. For instance, $|\{partsupp\}| = |\{supplier, partsupp\}|$. This effect is due to the fact that `partsupp` is a core relation of the query (see Fig. 6). For example, consider two views, {`partsupp`} and {`supplier, partsupp`}. Since there is an RI join edge from `partsupp` to `supplier`, the join can only "append" more values from `supplier` to each tuple in `partsupp`. Since a key of `partsupp` (`partkey` and `suppkey`) is among the query's output attributes, these appended values will not change the number of distinct tuples in the views. In general, we have the following result:

Theorem 5.1 *For any relation R in query Q, let V be the view associated with $\{R\}$. For any view $V' = \{R, S_1, \ldots, S_k\}$, such that each relation S_i is on a directed RI-join path from R, the number of distinct tuples in V is the same as in V': $|V| = |V'|$.* □

From this theorem it follows that we can use the number of distinct tuples in the view V to obtain the number of distinct tuples in the views V'. This theorem can help us reduce the amount of effort in estimating the number of distinct tuples in views, sometimes dramatically. For instance, for the query Q_2, after ignoring the `region` relation when enumerating relation subsets, we only need to estimate the number of distinct tuples in four views: {`nation`}, {`supplier`}, {`partsupp`}, and {`part`}. For any other view we consider for the query, its number of tuples can be found from these four numbers.

5.2 Estimating the Number of Distinct Tuples in a View

To estimate the number of distinct tuples in a view, we treat all values in a tuple, taken together, as a single value. In this way, we can reduce this problem to that of estimating the number of distinct values of a specific attribute in a relation. This problem has received a lot of attention in query optimization. In our approach, we focus on two estimators, namely the Smoothed Jackknife Estimator and Shlosser's Estimator [15]. Their details are in the Appendix.

5.3 Using the Client-Site Cache

Sometimes the client may have access to the cached results of previous queries. These results can help us reduce the communication costs of sending data from the server to the client. Suppose one view in a decomposition can be computed using previously cached results. Then the server does not need to transfer to the client the answer to that view. Thus, the view size should be 0 in our computation of decomposition plans, and the answer to the view will not be computed. For views whose answer is not sent from the server, the client needs to use its cached data to calculate the view's answer before computing the final answer to the query. In order to use cached data to reduce the communication costs, the server needs to have only a description (i.e., metadata) of the cached data, i.e., the server does not need to have access to the data itself.

6 Experiments

In this section we present the results of our extensive experiments and show that our proposed approach is efficient and effective when applied to representative large-scale applications.

6.1 The Queries and the Setting

To choose a set of realistic queries with reasonably large result sizes, we used slight modifications of the queries in the TPC-H benchmark [24]. The benchmark queries are representatives of typical queries in a wide range of decision-support applications. The TPC-H database schema contains various data types, and the queries have considerable redundancies. We will see in this section that our approach is effective in removing the redundancies. We removed all the grouping and aggregation operations from the queries. We also did some other minor modifications, such as adding more selection conditions and adding or removing some attributes in the outputs. The outcome of these modifications is a set of realistic queries, with result sizes varying between 1MB and 20MB.

To obtain the answers to the modified queries, we used the databases in the TPC-H benchmark. We experimented with databases of several sizes by varying the scaling factor. We have generated a total of 19 queries and applied our algorithm to them. Since our observations were consistent across the queries, for brevity we report the results for four queries only, namely q_1, q_2, q_3, and q_4. (The definitions of the modified queries are given in the Appendix.) We used Microsoft SQL Server Version $8.00.294$. The program (implemented in C++) interacts with the database using the standard ODBC interface. To simulate the client-server environment, we ran the experiments on two desktop PC's, for the server and the client respectively. The server machine has a Pentium IV 1.6GHz CPU, 256MB memory, and a 80GB hard disk. The client machine has a AMD Atholon XP 1.47GHz CPU, 256MB memory, and a 40GB hard disk.

6.2 Estimating View Sizes

In this section we report the results on the accuracy of our view-size estimators. We have implemented both the Smoothed Jackknife Estimator and the Shlosser's Estimator [15]. We chose the Smoothed Jackknife estimator since it returns more accurate results for our dataset and queries than the other one. In doing the estimation for each query, we typically sampled $8,000$ tuples in the answer to the extended query $\hat{Q}$, except two cases. (1) If the answer to $\hat{Q}$ had less than $8,000$ tuples, we used the whole answer, because in this case, the computing time was negligible. (2) If the size of the answer to $\hat{Q}$ was very large, say, 5% of the whole size is larger than 8000, we sampled 5% of the tuples in the results. We ran the four queries on different database sizes. Figure 7 shows how the accuracy changed for different views for queries q_3 and q_4. The formula for accuracy is $1 - |(size_{estimate} - size_{real}) \ / \ size_{real}|$. The experiments corroborated good accuracy of the estimators. In most cases, accuracy was close to 90%.

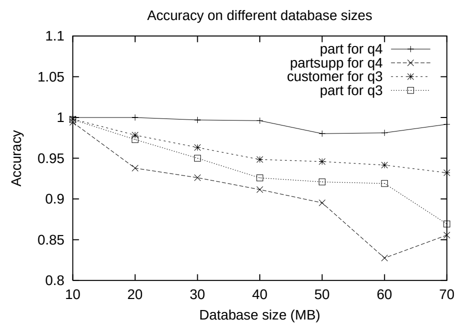


Figure 7. Accuracy on varying data sizes.

6.3 Pruning: The Number of Plans

To see how effectively the pruning rules in Section 4 can help the algorithm reduce the number of plans in the search space, we applied the rules to the queries. Table 3 shows the results. The experiments showed that using the three rules can result in a significant reduction in the size of the search space.

6.4 Reduction of Data-Communication Costs

We evaluated how much the algorithm can reduce the size of the query answers by comparing our approach (denoted as "DECOMP") with the naive approach that transfers the entire query answer ("NAIVE"). To measure the effect of size reduction, we introduce a measurement of reduction ratio for an approach A, as the ratio of data size of the

Table 3. Effect of pruning rules.

Queries	Original number of plans	After pruning rule 1	After pruning rule 2	After pruning rule 3
q_1	203	32	32	4
q_2	15	8	8	8
q_3	15	8	8	2
q_4	15	8	8	4

Table 5. Communication-reduction ratio.

DB size	q_1	q_2	q_3	q_4
10MB	1.73	5.79	2.71	2.64
20MB	1.7	5.84	2.78	2.71
30MB	1.39	5.79	2.76	2.68
40MB	1.38	5.91	2.76	2.67
50MB	1.38	5.94	2.75	2.66
60MB	1.36	5.88	3.36	2.64
70MB	1.38	5.9	2.78	2.64
80MB	1.35	5.94	2.72	2.63

NAIVE approach over that of approach A. That is:

$$\text{reduction ratio of approach } A = \frac{\text{data size of NAIVE}}{\text{data size of } A}$$

Here the approach A could be our DECOMP approach, or other approaches we will talk later, such as the compression approach, and the combinations of these two.

Table 4 shows the results on a 60MB-database (the scaling factor of 0.06); it can be seen that our technique dramatically reduced the data-communication costs for these queries. Take query q_2 as an example. The original size of the answer to the query (i.e., the size of NAIVE) was 11.4 MB. After applying our technique, we reduced the size to 1.94 MB, which was only 17% of the NAIVE size. That is, the reduction ratio was 5.88.

Table 4. Data-communication size.

Queries	NAIVE (MB)	DECOMP (MB)	Reduction ratio
q_1	3.45	2.53	1.36
q_2	11.4	1.94	5.88
q_3	14.3	4.26	3.36
q_4	13.7	5.19	2.64

Table 5 shows the data-communication-reduction ratio for the queries on several databases of different sizes. For example, on an underlying database of 30MB, the reduction ratio for q_1 was 1.39. That is, the size of data transferred by the NAIVE approach was 1.39 times of that of our DECOMP approach. Similarly, the reduction ratio was 5.79 for q_2, 2.76 for q_3, and 2.68 for q_4. Our experiments show that the technique can reduce the data-communication costs on databases of various sizes.

6.5 Running Time

In this subsection, we show the query-runtime reduction when using our approach. We consider both the transmission time and total running time on the server and client. The total running time includes (1) the time it takes to run the extended query, (2) the time for generating an optimal plan (including the time of searching for the optimal plan and the time for computing its views), (3) the transmission time, and (4) the time for the client-site execution. In our implementation, we computed the results of the views in the optimal plan by running the corresponding view queries in the database, since this approach was much more efficient than doing projections in the program. (The reason is that the database has indexing structures to do the queries efficiently.)

On the client site, because of the lack of indexing structures, we used a merge join algorithm to join the views to compute the final answer. To speed up the client-site join, on the server we tried to make the view results sorted based on the join attributes whenever it was possible. (This observation was called "interesting order" in System R [21]). Later we might still need to do some sorting if the join attributes for the merge-sort join was different from those already sorted. In this way, we reduced the time of the client-site join dramatically compared to the transmission time.

6.5.1 Different Running Times

Figure 8 shows how the different times in our approach vary for different database sizes. Query q_2 was chosen under a fixed network bandwidth of 56KB/sec. We can see that as the database size increases, the network transmission time dominates the total running time. Although other times also increase, their growth is relatively slow compared to the transmission time. Thus our approach would have a big advantage over the NAIVE approach when a large size of query result needs to be transfered via a very slow network.

Now we compare our DECOMP approach with the NAIVE approach. Experiments showed that our approach performs better than NAIVE in most cases, especially when the network is slow. (Not surprisingly, as the network bandwidth becomes higher, the advantage of our approach becomes smaller.)

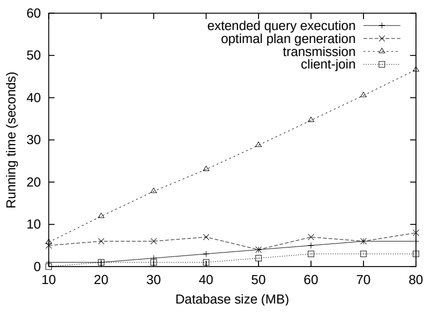


Figure 8. Runtime of q_2 on different data sizes.

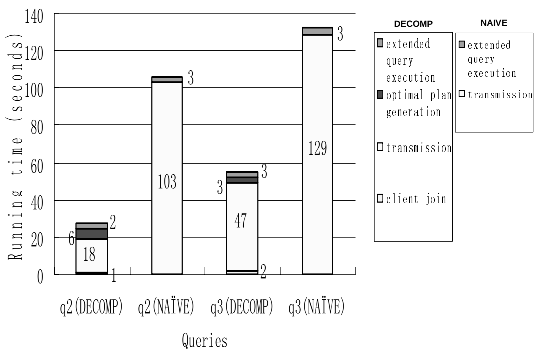


Figure 9. Runtime of DECOMP and NAIVE.

6.5.2 Total Running Time

Figure 9 compares the running time of DECOMP and NAIVE on queries q_2 and q_3, on a 30 MB database with the bandwidth of 56 KB/sec. Each bar for our approach has four blocks, corresponding to the four steps (from the top to the bottom). Each bars for the NAIVE approach has only two blocks, corresponding to the original-query execution and transmission. Since some of the times are too small to be displayed, we also put the number beside each block.

The figure shows that, although DECOMP takes two extra steps (optimal-plan generation and client-site execution), its total time is much smaller than the NAIVE DECOMP approach, due to the low bandwidth. The time for the two extra steps can pay off compared to the network transmission time, especially when the network becomes a bottleneck in the computation. In addition, even though NAIVE executed the original query, and DECOMP executed an extended query, the time difference was very small. Furthermore, the client-site join took relatively little time.

Under the same bandwidth of 56KB/sec, Figure 10 shows the total running time on different data sizes for queries q_2 and q_3. It shows that our DECOMP takes less time than the NAIVE approach, as the data size varies.

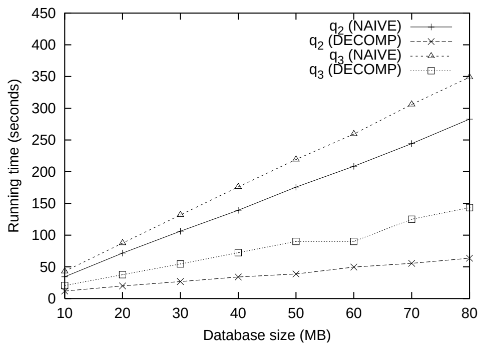


Figure 10. Total running time of NAIVE and DECOMP on different data sizes.

6.5.3 Running Time and Query Size

Table 6 illustrates the effect of network bandwidth on the benefits of our approach. It shows the amount of transfer time our approach can save over NAIVE, for different network bandwidths on a fixed database size of 60MB. We can see that, as expected, the advantage becomes smaller as the network becomes faster.

6.5.4 Running Time and Network Bandwidth

Figure 11 illustrates the effect of bandwidth on the benefits from our approach. It shows the total running time of DECOMP and NAIVE, for different network bandwidths on a fixed database size of 60MB. (Notice that the y-axis is using logarithmic scale.) Here we only show the result for queries q_2 and q_3. From the figure, we can see that our DECOMP approach saves time significantly under low network bandwidth. In particular, our approach behaves much better than NAIVE approach when the bandwidth is lower than 900 KB/sec, which is true for many network settings. For instance, for query q_2, when the bandwidth was 56KB/sec, the NAIVE approach took 209 seconds, while our approach took only 50 seconds, saving 159 seconds.

6.6 Comparison with Data Compression

There are other approaches to achieving the goal of minimizing the communication cost for transferring query results by reducing the data. Data compression is one of them [7]. Our approach is orthogonal to the compression approach, since the two approaches could be combined to achieve a better improvement. Now we report our experimental results to verify this claim.

In a compression approach, the result of a query is compressed before the transmission. On the client site, the client need to decompress the data to get the original result. So it includes four steps: executing the query, compressing the

Table 6. Transfer time saving (seconds) of DECOMP over NAIVE.

Bandwidth (KB/sec)	1	2	4	8	16	32	64	128	256
q_2	9450	4720	2355	1173	581	286	138	64	27
q_3	10030	5010	2500	1245	618	304	147	68	29

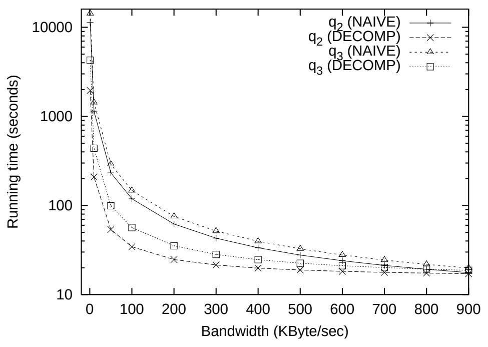


Figure 11. Running time of NAIVE and DECOMP on different bandwidths.

Table 7. Reduction ratio for different queries.

Queries	Reduction ratio (DECOMP)	Reduction ratio (ZIP)	Reduction ratio (DECOMP&ZIP)
q_1	1.36	4.3	3.46
q_2	5.88	7.08	14.16
q_3	3.36	3.4	12.12
q_4	2.64	4.27	9.51

Table 8. Reduction ratio of q_3 on different data sizes.

DB size (MB)	Reduction ratio (DECOMP)	Reduction ratio (ZIP)	Reduction ratio (DECOMP&ZIP)
10	2.71	3.49	9.96
20	2.78	3.49	10
30	2.76	3.45	9.96
40	2.76	3.44	9.93
50	2.75	3.42	10.17
60	3.36	3.4	12.12
70	2.73	3.4	10.06
80	2.72	3.4	10.05

result, transmitting the result, and decompressing the data. Clearly we can utilize both our approach and the compression. For instance, we can first decompose queries into subqueries using our DECOMP approach, and further reduce the data size by compressing the results of views. Even though the redundancy could be smaller after our approach, the compression step can still achieve a good reduction ratio, as shown by our experiments. As a consequence, when combining the two approaches, we gain more time saving to transmit query results.

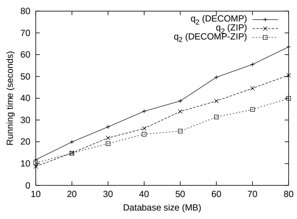
We conducted experiments to verify our analysis. There are many compression tools, and we chose WinZip [26] due to its availability and good efficiency. We collected the running time and data-size-reduction ratio of our DECOMP approach, the WinZip approach ("ZIP"), and their combination ("DECOMP&ZIP").

Table 7 lists the reduction ratio for the three approaches on different queries with fixed database size of 60MB. It shows that by combining DECOMP and COMP, we gain much higher reduction ratio. For example, for query q_3, the ratio for DECOMP was 3.36, the ratio for ZIP was 3.4, and the ratio of their combination DECOMP&ZIP was 12.12.

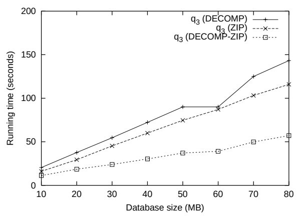
Table 8 shows the reduction ratio for different approaches on different database sizes, for query q_3. Again, we can see that in most cases DECOMP&ZIP has much higher reduction ratio over DECOMP and ZIP. For instance, when the database size was 60MB, the ratios of DECOMP, ZIP, and DECOMP&ZIP were 3.36, 3.4, and 12.12, respectively.

Figure 12 shows the total running time of query q_2 and q_3 on different data sizes with the fixed bandwidth of 56KB/sec. The total running time includes all the times of different steps for each approach. Among the three approaches, DECOMP&ZIP always needs the smallest amount of time. The reason is that DECOMP&ZIP always gets the highest reduction ratio, thus has smallest amount of data to be transferred, and when the network is the bottleneck, it can save much time. For instance, for query q_3, when the data size was 70 MB, DECOMP took 125 seconds, ZIP took 103 seconds, and DECOMP&ZIP took only 50 seconds.

Now we fix the database size to 60MB, and let the bandwidth vary. Figure 13 shows the total running time for the three approaches. As expected, DECOMP&ZIP is a winner when the bandwidth is low. When the bandwidth was 20KB/sec, DECOMP&ZIP saved 33 seconds over ZIP for query q_2, and 145 seconds for query q_3. As expected, this benefit becomes less as the network speed goes down.

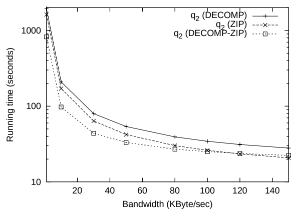


(a) Query q_2

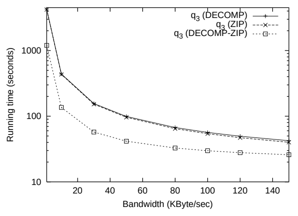


(b) Query q_3

Figure 12. Running time for DECOMP, ZIP, and DECOMP&ZIP on different data sizes.



(a) Query q_2



(b) Query q_3

Figure 13. Running time for DECOMP, ZIP, and DECOMP&ZIP on different bandwidths.

Summary

Our extensive experiments have shown that our proposed approach can reduce the data-communication size significantly when the query result has a lot of redundancy. This reduction can save the total running time of sending the query result to the client, especially when the network bandwidth is low. Even though our approach needs to compute an extended query, search for an optimal plan to do the decomposition, and generate the results of different views, these extra steps can pay off since they can reduce the data size. In addition, our approach is orthogonal to the data-compression technique, and combining these two can further reduce the communication costs, and thus save the total running time to send the query result.

7 Conclusions

In this paper we studied the problem of minimizing the communication costs of transferring the answer to a large-join query from a server to a client; the problem exists in a variety of database applications. We proposed a novel technique that decomposes queries into intermediate results, called "views"; the answers to the views are transferred to the client and are then used by the client to compute the answers to the queries. Decomposing queries into views can reduce the redundancy in the query answers, which may result in significant reductions in the costs of transferring the data from the server to the client. We discussed several challenges we had addressed in implementing this technique. In our extensive experiments, we used queries that are modifications of TPC-H benchmark queries. The experiments show that our technique can significantly reduce the communication costs of transferring the answers to large-join queries from server to client.

Currently we are extending the work reported in this paper, by taking into account the computing power of the client and of the server. The focus of our investigation is on the tradeoffs between minimizing the communication costs and reducing the computation load on both the client and the server. Moreover, the computation of query answers can be pipelined at the client as view results arrive. By studying these issues, we expect to come up with an even more powerful technique, which would maximize the overall performance of answering large-join queries.

References

[1] S. Agrawal, S. Chaudhuri, and V. Narasayya. Automated selection of materialized views and indexes in Microsoft SQL Server. In *Proc. of VLDB*, pages 496–505, Cairo, Egypt, 2000.

[2] P. M. G. Apers, A. R. Hevner, and S. B. Yao. Optimization algorithms for distributed queries. *IEEE Transactions on Software Engineering (TSE)*, 9(1):57–68, 1983.

[3] T. Barsalou, A. M. Keller, N. Siambela, and G. Wiederhold. Updating relational databases through object-based views. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 248–257, 1991.

[4] P. A. Bernstein and D.-M. W. Chiu. Using semi-joins to solve relational queries. *JACM*, 28(1):25–40, 1981.

[5] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems*. McGraw-Hill Book Company, 1984.

[6] M.-S. Chen and P. S. Yu. Combining join and semi-join operations for distributed query processing. *TKDE*, 5(3):534–542, 1993.

[7] Z. Chen and P. Seshadri. An algebraic compression framework for query results. In *ICDE*, pages 177–188, 2000.

[8] R. Chirkova and M. R. Genesereth. Linearly bounded reformulations of conjunctive databases. *DOOD*, 2000.

[9] R. Chirkova, A. Y. Halevy, and D. Suciu. A formal perspective on the view selection problem. *Proc. of VLDB*, pages 59–68, 2001.

[10] R. Chirkova and C. Li. Materializing views with minimal size to answer queries. *PODS*, 2003.

[11] D.Kossmann and M.Franklin. A study of query execution strategies for client-server database systems. Technical report, Univ. Maryland, 1995.

[12] S. J. Finkelstein. Common subexpression analysis in database applications. In *SIGMOD*, pages 235–245, 1982.

[13] M. Graham. On the universal relation. Technical report, University of Toronto, Canada, 1979.

[14] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proc. of VLDB*, pages 276–285, 1997.

[15] P. J. Haas, J. F. Naughton, S. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *Proc. of VLDB*, pages 311–322, 1995.

[16] H. Hacıgümüş, B. Iyer, and S. Mehrotra. Providing database as a service. In *ICDE*, 2002.

[17] B. S. Lee and G. Wiederhold. Efficiently instantiating view-objects from remote relational databases. *VLDB Journal: Very Large Data Bases*, 3(3):289–323, July 1994.

[18] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *Proceedings VLDB'86*, pages 149–159, 1986.

[19] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, 1999.

[20] M. T. Roth, F. Ozcan, and L. M. Haas. Cost models DO matter: Providing cost information for diverse data sources in a federated system. In *The VLDB Journal*, pages 599–610, 1999.

[21] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.

[22] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *VLDB Journal*, 5(1):48–63, 1996.

[23] J. A. Storer. *Data Compression: Methods and Theory*. Computer Science Press, 1988.

[24] Transaction Processing Performance Concil: TPC. `http://www.tpc.org/`.

[25] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, 1992.

[26] WinZip. http://www.winzip.com/.

[27] M. M. Zloof. Query-by-Example: A data base language. *IBM Systems Journal*, 16(4):324–343, 1977.

A Appendix

A.1 Estimators for the Number of Distinct Values

In this subsection we describe two estimators, namely the Smoothed Jackknife Estimator, and the Shlosser's Estimator, to estimate the number of distinct values of a specific attribute in a relation. These estimators are used in our experiments to esimate the size of each view. See [15] for more details about the estimators.

Without loss of generality, we consider a relation R having N tuples, and one of its attribute having D distinct values, numbered $1, 2, \ldots, D$. A sample of n tuples selected randomly from R is used for estimation, and the number of distinct attribute values in the sample is denoted as d. Also we denote by $\overline{N}$ the average number of tuples for an attribute value, i.e., $\overline{N} = N/D$. We denote by N_j the number of tuples in R with attribute value j, for $1 \le j \le D$. For $1 \le i \le n$, let f_i be the number of attribute values that appear exactly i times in the sample.

The Smoothed Jackknife Estimator

This estimator is based on correcting bias of d_n, that is $E[d_n] - D$, where $E[d_n]$ is the expectation of d_n. Here we write $d = d_n$, and use d_{n-1} to stand for the average number of distinct attribute values in the sample after one tuple is removed. d_{n-1} can be calculated by:

$$d_{n-1} = \frac{\sum_{k=1}^{n} d_{n-1}(k)}{n}$$

where $d_{n-1}(k)$ means the number of distinct attribute values in the sample after the k'th tuple is removed from the sample, for $1 \le k \le n$. In addition, the estimator uses the following terms.

- The squared coefficient of variation of the frequencies $N_1, N_2, \ldots, N_D$:

$$\gamma^2 = \frac{\frac{1}{D}\sum_{j=1}^{D}(N_j - \overline{N})^2}{\overline{N}^2}$$

- The probability that the attribute value j does not appear in the sample.

$$h_n(N_j) = \frac{\Gamma(N - N_j + 1)\Gamma(N - n + 1)}{\Gamma(N - n - N_j + 1)\Gamma(N + 1)}$$

  Here $\Gamma(x+1) = x!$ for nonegative integer x.

Based on this estimator, the estimated number of distinct values in R, denoted by $\hat{D}_{CJ}$, is:

$$\hat{D}_{sjack} = d_n - K(d_{n-1} - d_n)$$

where K is:

$$K \approx -\Big(\frac{N - \overline{N} - n + 1}{\overline{N}}\big(1 - \frac{\overline{N}\gamma^2 h'_{n-1}(\overline{N})}{h_{n-1}(\overline{N})}\big)\Big)$$

and h'_{n-1} in K is the first derivative of h_{n-1}.

The Shlosser's Estimator

Based on this estimator, the estimated number of distinct values in R, denoted by $\hat{D}_{Shloss}$, is:

$$\hat{D}_{Shloss} = d + \frac{f_1 \sum_{i=1}^{n}(1-q)^i f_i}{\sum_{i=1}^{n} i \times q(1-q)^{i-1} f_i}$$

based on the assumption that each tuple is included in the sample with probability $q = n/N$, independently of all other tuples. It performs well when the attribute value distribution is not uniform.

A.2 Sample Queries Used in the Experiments

Our experiments used the following queries, which are adapted from the TPC-H benchmark. We removed all the grouping and aggregation operations from the queries. We also did other minor modifications, such as adding more selection conditions and adding or removing some attributes in the `SELECT` clause..

```
q1:
SELECT  l_extendedprice, l_discount,
        l_quantity, l_orderkey,
        l_lineitem, n_name as nation,
        year(o_orderdate) as o_year,
        ps_supplycost
FROM    lineitem, nation, orders, part,
        partsupp, supplier
WHERE   l_quantity > 30
        AND l_discount > 0.05
        AND s_suppkey = l_suppkey
        AND ps_suppkey = l_suppkey
        AND ps_partkey = l_partkey
        AND p_partkey = l_partkey
        AND o_orderkey = l_orderkey
        AND s_nationkey = n_nationkey;

q2:
SELECT  c_custkey, c_name, c_acctbal,
        c_address, c_phone, c_comment,
        l_extendedprice, l_discount,
        l_orderkey, l_lineitem, n_name
FROM    customer, lineitem,
        nation, orders
WHERE   c_custkey = o_custkey
        AND c_nationkey = n_nationkey
        AND l_orderkey = o_orderkey
        AND o_orderdate >= '1997-06-01'
        AND l_returnflag = 'N';

q3:
SELECT  c_custkey, c_name, c_address,
        c_phone, c_acctbal, c_comment,
        l_quantity, l_extendedprice,
        l_orderkey, l_lineitem,
        o_orderkey, o_orderpriority,
        o_clerk, p_name, p_brand,
        p_type, p_comment
FROM    customer, orders, lineitem,
        part
WHERE   c_custkey = o_custkey
        AND o_orderkey = l_orderkey
        AND l_partkey = p_partkey
        AND (l_quantity < 10)
        AND (l_discount <= 0.08);

q4:
SELECT  l_partkey, l_linenumber,
        l_discount, l_comment,
        l_orderkey, p_name, p_mfgr,
        p_comment, ps_suppkey,
        ps_availqty, ps_supplycost,
        ps_comment, s_name
FROM    lineitem, part, partsupp,
        supplier
WHERE   s_suppkey = ps_suppkey
        AND p_partkey = ps_partkey
        AND ps_suppkey = l_suppkey
        AND ps_partkey = l_partkey
        AND l_discount < 0.04
        AND p_size < 20;

q5:
select  s_acctbal, s_name, n_name,
        p_partkey, p_mfgr, s_address,
        s_phone, s_comment,
        ps_supplycost
from    part, supplier,
        partsupp, nation, region
where
        p_partkey = ps_partkey
        and s_suppkey = ps_suppkey
        and p_type like '%TIN'
        and s_nationkey = n_nationkey
        and n_regionkey = r_regionkey
        and r_name = 'ASIA'

q6:
select  l_orderkey,
        l_extendedprice*(1-l_discount))
          as revenue,
        o_orderpriority, o_orderdate,
        o_shippriority
from    customer, orders, lineitem
where   c_mktsegment = 'BUILDING'
        and c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and l_shipdate > '1992-03-15'

q7:
select  c_name, c_phone, n_name,
        o_totoalprice, o_comment,
        l_extendedprice, l_discount
from    customer, orders, lineitem,
        supplier, nation, region
where   c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and l_suppkey = s_suppkey
        and c_nationkey = s_nationkey
        and s_nationkey = n_nationkey
        and n_regionkey = r_regionkey
        and r_name = ’ASIA’
        and o_orderdate >=’1991-01-01’

q8:
select  year(o_orderdate) as o_year,
        o_orderstatus, o_totalprice
        o_comment, s_name, s_phone,
        n2.n_name as nation
        l_extendedprice*(1-l_discount)
          as volume,
from    part, supplier, lineitem,
        orders, customer,
        nation n1, nation n2, region
where   p_partkey = l_partkey
        and s_suppkey = l_suppkey
        and l_orderkey = o_orderkey
        and o_custkey = c_custkey
        and c_nationkey = n1.n_nationkey
        and n1.n_regionkey = r_regionkey
        and r_name = ’EUROPE’
        and s_nationkey = n2.n_nationkey
        and o_orderdate >= ’1995-01-01’
        and o_orderdate <= ’1996-12-31’
        and p_type =’LARGE PLATED BRASS’

q9:
select  ps_partkey, ps_supplycost,
        ps_availqty, s_name, s_address,
        s_acctbal,n_name
from    partsupp, supplier, nation
where   ps_suppkey = s_suppkey
        and s_nationkey = n_nationkey
        and n_name = ’CANADA’;

q10:
select  l_shipmode, l_receiptdate,
        o_totalprice, o_shippriority,
        o_orderpriority, o_comment
from    orders, lineitem
where   o_orderkey = l_orderkey
        and l_shipmode in
          (’TRUCK’, ’REG AIR’)
        and l_commitdate < l_receiptdate
        and l_shipdate < l_commitdate
        and l_receiptdate>=’1996-01-01’

q11:
select  c_name, c_custkey, o_orderkey
from    customer, orders
where   c_custkey = o_custkey
        and o_comment not like
        ’\%pending\%requests\%’;

q12:
select  p_type, p_name, p_mfgr,
        p_retailprice, l_extendedprice,
        l_discount
from    lineitem, part
where   l_partkey = p_partkey
        and l_shipdate >= ’1993-09-01’

q13:
select  s_suppkey, s_name, s_phone,
        l_suppkey, l_extendedprice,
        l_discount, l_shipdate
from    supplier, lineitem
where   s_suppkey = l_suppkey
        l_shipdate >= ’1996-01-01’

q14:
select  p_brand, p_type, p_size,
        ps_suppkey
from    partsupp, part
where   p_partkey = ps_partkey
        and p_brand <> ’Brand#15’
        and p_type not like
          ’SMALL POLISHED%’

q15:
select  p_name, p_mfgr, p_type,
        l_extendedprice, l_quantity
from    lineitem, part
where   p_partkey = l_partkey
        and p_brand = ’Brand#42’
        and p_container = ’SM CASE’;

q16:
select  c_name, c_address, c_custkey,
        o_orderkey, o_orderdate,
        o_totalprice, l_quantity
from    customer, orders, lineitem
where   c_custkey = o_custkey
        and o_orderkey = l_orderkey;

q17:
select  p_name, p_brand, l_discount,
        l_extendedprice,l_shipinstruct
```

```
from    lineitem, part
where   p_partkey = l_partkey
        and l_quantity >= 2
        and l_shipmode in
          ('AIR', 'AIR REG')


q18:
select  s_name, s_address, s_suppkey,
        p_partkey, ps_partkey,
        ps_availqty, l_quantity
from    supplier, nation, part,
        lineitem, partsupp
where   p_name not like 'beige%'
        and s_nationkey = n_nationkey
        and n_name = 'MOROCCO'
        and s_suppkey = ps_supkey
        and ps_parktey = p_partkey
        and l_partkey = ps_partkey
        and l_suppkey = ps_suppkey
        and l_shipdate >= '1996-01-01'
        and l_shipdate < '1999-01-01'


q19:
select  s_name, s_address, o_orderdate,
        o_orderpriority, l_orderkey,
        l_suppkey, l_receiptdate,
        l_commitdate
from    supplier, lineitem, orders,
        nation
where   s_suppkey = l_suppkey
        and o_orderkey = l_orderkey
        and o_orderstatus = 'F'
        and l_receiptdate > l_commitdate
        and s_nationkey = n_nationkey
        and n_name = 'KENYA'
```