Materializing Views with Minimum Size to Answer Queries
(Technical Report)

Rada Chirkova  
North Carolina State University  
chirkova@csc.ncsu.edu  

Chen Li  
UC Irvine  
chenli@ics.uci.edu  

Jia Li  
UC Irvine  
jiali@ics.uci.edu  

May 5, 2004

Abstract

In this paper we study the following problem. Given a database and a set of queries, we want to find a set of views that can compute the answers to the queries, such that the size of the views (i.e., the amount of space, in bytes, required to store the views) is minimal on the given database. This problem is important for applications such as distributed databases, data warehousing, and data integration. We explore the decidability and complexity of the problem for workloads of conjunctive queries. We show that results differ significantly depending on whether the workload queries have self-joins. Further, for queries without self-joins we describe a very compact search space of views, which contains at least one optimal view set. We apply our theoretical results using a real-life application: In a client-server framework, we devise an efficient practical approach that finds a provably optimal solution to the problem of minimizing the communication costs of transferring answers to large-join queries when the answers have a lot of redundancy. We validate the approach by extensive experiments and discuss competing or complementary approaches.

Keywords: views, data warehouses, distributed systems, client-server systems, minimizing communication costs

1 Introduction

In this paper we study the following problem: given a set of queries, how to choose views to compute the answers to the queries, such that the total size of the view set (i.e., the amount of space, in bytes, required to store the view set) is minimal. This problem exists in many environments, such as distributed databases [BGW81, CP84, ÖV99], data integration [Len02], and the recent “database-as-a-service” model [HILM02]. For example, mediators in data-integration systems support seamless access to autonomous, heterogeneous information sources [Wie92]. A mediator translates a given user 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 [HKWY97]. After receiving many user queries, the mediator can send multiple queries to the same source to receive data. As another example, “database as a service” is a new model for enterprise computing [HILM02], in which companies and organizations choose storing their data on a server over having to maintain local databases. The server provides client users with the power 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 answer and sends the results to the user over the network.

These applications share the following characteristics. (1) Both the client and the server are able to do computation. Notice that 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; in the mediation scenario, the client (mediator) might have to do computation anyway. (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, and the client may want to minimize the costs of transferring the data over the network.

In client-server applications, an important metric is minimizing data-transfer time. In contrast, in a data warehouse, minimizing data-transfer time is irrelevant, but it is crucial to keep around materialized views whose total size is small as possible, to try to prevent accessing original stored relations in the data sources when processing and answering warehouse queries. Here the problem is how to materialize locally (in the warehouse) relations with small total size, to avoid excessive transfer time for big original stored relations.

In general, given a set of queries (a query workload) and a fixed database instance, we want to define and compute a set of intermediate results (views), such that these view results can be used to compute the answer to each query in the workload. In addition, we want to choose the views in such a way that their total size is minimum on the given database. In this paper we study this problem for select-project-join queries. We make the following contributions:

1. In Section 3 we study the decidability and complexity of the problem. We show that if workload queries have self-joins, nontrivial disjunctive views can give rise to smaller viewsets than purely conjunctive views. We establish that the problem of finding a minimum-size viewset in the space of disjunctive views is decidable, and give an upper bound on the complexity of the problem. Further, we show that for arbitrary conjunctive query workloads, to find rewritings of the workload queries in terms of a minimum-size viewset it is not necessary to consider nontrivial disjunctive rewritings.

2. In Section 4 we study workloads of conjunctive queries without self-joins, and show that disjunctive views cannot provide smaller viewsets than purely conjunctive views. Thus it is enough to consider purely conjunctive views when looking for a minimum-size disjunctive viewset. Moreover, it is enough to explore a very restricted search space of such views, and the problem of finding a minimum-size viewset is in NP.

3. In Section 5, we study how to apply these results in client-server environments. We consider the case of a single query without self-joins, and the problem becomes how to decompose query answers into intermediate results (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. We study how to find a provably globally optimal solution efficiently, and our approach can be used on top of the server database-management system.

4. In Section 6 we present our experimental results, which show that our proposed approach can reduce the data-communication size significantly when the query result has a lot of redundancy.

1.1 Related Work

The problem of finding views to materialize to answer queries has traditionally been studied under the name of view selection. Its original motivation comes up in the context of data warehousing. The problem is to decide which views to store in the warehouse to obtain optimal performance [Gup97, TLS99, TS97, YKL97]; one direction is to materialize views and indexes for data
cubes in online analytic processing (OLAP) [BPT97, GHRU97, HRU96]. Another motivation for view selection is provided by recent versions of several commercial database systems. These systems support incremental updates of materialized views and are able to use materialized views to speed up query evaluation [BDD98, GL01, ZCL+00]. Choosing an appropriate set of views to materialize in the database is crucial in order to obtain performance benefits from these new features [ACN00].

Traditional work on view selection uses certain critical tacit assumptions. The first assumption is that the only views to be considered to materialize are those that are subexpressions of the given queries, or are given in the input to the problem in some other way. The second assumption is that there is some low upper bound on the number of views in an optimal viewset. These assumptions have been questioned in recent work on database restructuring [Chi02, CG00, CHS01], which considers all possible views that can be invented to optimize a given metric of database performance.

Other related topics include answering queries using views (e.g., [ALU01, LMSS95, Hal01]), view-based query answering (e.g., [CGL00, CGLV00]), and minimizing viewsets without losing query-answering power [LBU01]. In addition, there has been a lot of work on minimizing data-communication costs in distributed database systems (e.g., [CP84, ÖV99]). Our novel approach complements existing techniques. In particular, we experimentally evaluated our approach and the data-compression approaches [CS00]. We show that our approach is orthogonal to the data-compression technique, and combining the two approaches in client-server systems can further reduce the communication costs and thus save on the total time required to send the query result to the client.

2 Problem Formulation

In this section we formulate the problem of finding a set of views with the minimum size to answer queries. We first present and discuss a motivating example and then give a formal specification of the problem.

2.1 Motivating Example

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

- customer(custkey(4), name(25), mktsegment(10))
- order(orderkey(4), custkey(4), orderdate(8), shippriority(4), comment(79))
- lineitem(linenum(4), orderkey(4), quantity(4), shipdate(8), shippriority(4), shipmode(10))

We underline the attribute(s) of the primary key of a 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 TPC-H benchmark [TPC]. The server computes the answer to $Q_1$ and sends it back to the client. Suppose there are 4000 tuples in the answer to $Q_1$ on the database at the server. It follows that the total number of bytes sent to the client is:

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

Figure 1: Query $Q_1$.

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>orderkey</th>
<th>comment</th>
<th>shippriority</th>
<th>orderdate</th>
<th>quantity</th>
<th>shipmode</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>Tom</td>
<td>134721</td>
<td>...</td>
<td>0</td>
<td>3/14/1995</td>
<td>26</td>
<td>REG AIR</td>
</tr>
<tr>
<td>$t_2$</td>
<td>Tom</td>
<td>134721</td>
<td>...</td>
<td>0</td>
<td>3/14/1995</td>
<td>75</td>
<td>REG AIR</td>
</tr>
<tr>
<td>$t_3$</td>
<td>Tom</td>
<td>134721</td>
<td>...</td>
<td>0</td>
<td>3/14/1995</td>
<td>43</td>
<td>AIR</td>
</tr>
<tr>
<td>$t_4$</td>
<td>Jack</td>
<td>571683</td>
<td>...</td>
<td>0</td>
<td>12/21/1994</td>
<td>43</td>
<td>MAIL</td>
</tr>
<tr>
<td>$t_5$</td>
<td>Jack</td>
<td>571683</td>
<td>...</td>
<td>0</td>
<td>12/21/1994</td>
<td>33</td>
<td>AIR</td>
</tr>
</tbody>
</table>

Table 1: Partial results of query $Q_1$.

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 necessary data to compute the final answer to $Q_1$. Table 1 shows a fragment of the answer to $Q_1$. 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, this information generates 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.

Figure 2: Decomposing the query answer into two 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 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.

Before giving the problem formulation, we first briefly review important concepts of conjunctive queries and answering queries using views.

2.2 Queries and Rewritings

We consider conjunctive queries in the form

$$\text{ans}(\bar{X}) ::= R_1(\bar{X}_1), \ldots, R_n(\bar{X}_n).$$

Predicate $R_i$ in a subgoal $R_i(\bar{X}_i)$ corresponds to a base (stored) relation, and each argument in the subgoal is either a variable or a constant. We consider views defined on base relations by safe conjunctive or disjunctive queries. A disjunctive query is a union of conjunctive queries. A query is safe if each variable in the query’s head appears in the body. A query variable is called distinguished if it appears in the query’s head. We assume set semantics for query answers.

For instance, the query $Q_1$ above can be represented as the following conjunctive query:

$$Q_1(N, OD, SP, C, OK, QT, SM) ::= \text{customer}(CK, N', BLDG'),$$
$$\text{orders}(OK, CK, OD, SP, C),$$
$$\text{lineitem}(LN, OK, QT, SD, SM).$$

A query $Q$ is contained in a query $Q'$, denoted $Q \sqsubseteq Q'$, if for any database $D$, the answer to $Q$ on $D$ is a subset of the answer to $Q'$ on $D$. The two queries are equivalent if $Q \sqsubseteq Q'$ and $Q' \sqsubseteq Q$. A conjunctive query $Q$ is contained in a conjunctive query $Q'$ if and only if there is a containment mapping from $Q'$ to $Q$ [CM77]. The expansion of a query $P$ on a set of views $\mathcal{V}$, denoted $P^{\exp}$, is obtained from $P$ by replacing all views in $P$ by their definitions in terms of the base relations. Given a query $Q$ and a set of views $\mathcal{V}$, a query $P$ is an equivalent rewriting of $Q$ using $\mathcal{V}$ if $P$ uses only the views in $\mathcal{V}$ and $P^{\exp}$ is equivalent to $Q$. In the rest of the paper, we use “rewriting” to mean “equivalent rewriting.”

For instance, the following are the views $V_1$ and $V_2$ from Section 2.1, in a conjunctive-query form.

$$V_1(N, OD, SP, C, OK) ::= \text{customer}(CK, N', BLDG'),$$
$$\text{orders}(OK, CK, OD, SP, C),$$
$$\text{lineitem}(LN, OK, QT, SD, SM).$$

$$V_2(OD, QT, SM) ::= \text{customer}(CK, N', BLDG'),$$
$$\text{orders}(OK, CK, OD, SP, C),$$
$$\text{lineitem}(LN, OK, QT, SD, SM).$$

The following query $P$ is an equivalent rewriting of the query $Q_1$ using the two views.
\[ P(N, OD, SP, C, OK, QT, SM) \leftarrow V_1(N, OD, SP, C, OK), V_2(OD, QT, SM). \]

In this paper we consider query rewritings that are either conjunctions of views (\textit{conjunctive} query rewritings) or unions of conjunctions of views (\textit{disjunctive} query rewritings), under set semantics.

2.3 Problem Statement

Given a set, or \textit{workload}, \( Q \) of conjunctive queries on stored relations \( R_1, \ldots, R_n \) and a fixed database instance \( D \), we want to find and precompute a set \( \mathcal{V} \) of intermediate results, defined as views \( V_1, \ldots, V_k \) on these relations, such that there exist equivalent rewritings to all the queries in the workload \( Q \) in terms of the views in \( \mathcal{V} \) only. Our goal is to find an \textit{optimal solution} — to choose, among all such sets of views \( \mathcal{V} \), a set \( \mathcal{V}^* \) whose total size

\[ \sum_{V_i \in \mathcal{V}^*} \text{size}(V_i) \]

is minimum on the given database instance. The size of a view \( V_i \), \( \text{size}(V_i) \), is the amount of space, in bytes, required to store the answer to \( V_i \) on the database \( D \). In addition to finding the views, we also find a plan to compute the answer to each query in the workload \( Q \) using the views in \( \mathcal{V} \).

3 Decidability and Complexity

In this section and in Section 4, we study decidability and complexity of the problem, and the search space of the possible solutions.

3.1 Different Types of Views

There are two types of views in a rewriting of a query: (1) containment-target views, and (2) filtering views [ALU01, PL00]. They can be distinguished by examining containment mappings from the query to the expansion of the rewriting. Intuitively, in a rewriting, a \textit{containment-target view} “covers” at least one query subgoal. Covering all query subgoals is enough to produce a rewriting of the query.

\textbf{Example 3.1} Consider two relations: \texttt{r(Dealer,Make)} and \texttt{s(Dealer,City)}. A tuple \texttt{r(d,m)} means that dealer \( d \) sells a car of make \( m \). A tuple \texttt{s(d,c)} means that dealer \( d \) is located in city \( c \). Consider the following query \( Q \) and three views \( V_1 \), \( V_2 \), and \( V_3 \). The query asks for all pairs \((m,c)\), such that there is a dealer in city \( c \) selling cars of make \( m \).

\[ Q : \ \text{ans}(M, C) \leftarrow r(D, M), s(D, C). \]
\[ V_1 : \ \text{ans}(D, M) \leftarrow r(D, M). \]
\[ V_2 : \ \text{ans}(D, C) \leftarrow r(D, M), s(D, C). \]
\[ V_3 : \ \text{ans}(D) \leftarrow r(D, M), s(D, C). \]

\( P \) is an equivalent rewriting of \( Q \) using the three views:

\[ P : \text{ans}(M, C) \leftarrow V_3(D), V_1(D, M), V_2(D, C). \]
We can show there are two containment mappings: one from $Q$ to the expansion $P^{exp}$ of $P$ — this mapping is an identity mapping — and another from $P^{exp}$ to $Q$.

View $V_1(D, M)$ covers the query subgoal $r(D, M)$, whereas view $V_2(D, C)$ covers the subgoal $s(D, C)$. Thus $V_1$ and $V_2$ are containment-target views for the query. □

**Definition 3.1** (Containment-target view) A conjunctive view $V$ is a containment-target view for a query $Q$ if the following is true. There exists a rewriting $P$ of $Q$ ($P$ uses $V$), and there is a containment mapping from $Q$ to the expansion $P^{exp}$ of $P$, such that $V$ provides the image of at least one subgoal of $Q$ under the mapping. □

A view is a filtering view for a query if it is not a containment-target view. Filtering views are not necessary in constructing query rewritings, in the sense that those views do not cover query subgoals. However, there could exist some query plan in which a filtering view removes — filters out — dangling tuples from some join input(s) in the plan, which may reduce the cost of evaluating the query. In Example 3.1, view $V_3$ is a filtering view, which removes from the view $V_1$ all tuples that are dangling with respect to the view $V_2$. We will show that to solve the problem of minimizing the size of a viewset that gives a rewriting of a query workload, we do not need to consider filtering views.

3.2 Minimum-Size Sets of Conjunctive Views: The Problem Is Decidable

We start by obtaining results on the decidability and complexity of finding a minimum-size conjunctive viewset for a set of conjunctive queries, assuming conjunctive rewritings. The length of a definition of a query (or view) is the number of subgoals in the query (view).

**Theorem 3.1** For any finite workload of conjunctive queries and a database instance, it is possible to construct a finite search space of views, which includes all views in at least one minimum-size conjunctive viewset for the workload. The number of views in the search space is at most doubly-exponential in the length of the longest query definition in the workload. □

**Proof:** We show that for any finite workload $Q$ of conjunctive queries and for any database instance $D$, there exists a minimum-size viewset $V$, such that each view $v$ in $V$ has at least one (conjunctive) definition whose length is at most singly-exponential in the length of the longest query definition in the workload $Q$. From this result, the claim of the theorem follows immediately.

Let $W$ be any minimum-size viewset for a workload $Q$ on a database instance $D$. Let $max_Q[|q|]$ be the length of the longest query definition in the workload $Q$, and let $size(W)$ be the total size of the relations (i.e., the number of bytes needed to store the relations) for the views in $W$ on the database $D$. Suppose some view in $W$ has a definition whose length is more than singly-exponential in the length of the longest query definition in the workload $Q$. By Theorem 3.1 in [CHS01], for $Q$, $D$, $W$, and for storage limit $size(W)$, there exists another viewset, $V$, with three properties: (1) for each view $v$ in $V$, the length of the definition of $v$ is at most singly-exponential in $max_Q[|q|]$, (2) the relations for the views in $V$ on the database $D$ satisfy the storage limit $size(W)$, and (3) for any query $q$ in $Q$, there is an equivalent rewriting of $q$ in terms of $V$. Moreover, by construction, there is a 1:1 mapping between the views in $W$ and the views in $V$, such that each view $v$ in $V$ is contained in the respective view $w$ in $W$. By construction, $V$ is a minimum-size viewset for the query workload $Q$ on the database $D$.

**Note 1.** We use the construction in the proof of Theorem 3.1 to observe that for any set of views $W$ that is a minimum-size viewset for the given $Q$ and $D$, there exists a set of views $V$ that is a
representative viewset of \( W \) in the space \( S \) of all sets of views whose definition length is at most singly-exponential in the length of the longest query definition in \( Q \), in the following sense. (1) \( \text{size}(V) = \text{size}(W) \) on \( D \), and (2) for each query \( q \) in \( Q \), a rewriting of \( q \) in terms of \( V \) is the result of replacing, in the rewriting of \( q \) in terms of \( W \), the literal for each view \( w \) in \( W \) by the literal for the image, in \( V \), of \( w \) under the 1:1 mapping from \( W \) to \( V \). Because the set \( S \) is finite, it takes finite time to construct all such representative viewsets of all minimum-size viewsets for a given \( Q \) and \( D \).

From Theorem 3.1 we obtain a decidability result:

**Corollary 3.1** Given a database instance, the problem of finding a minimum-size conjunctive viewset is decidable for finite workloads of conjunctive queries, assuming all rewritings are conjunctive.

**Note 2.** We observe that the problem has a triply-exponential upper bound: A naive algorithm will find a minimum-size viewset for a given query workload and database instance, by exploring all subsets of the at most doubly-exponential search space of views.

### 3.3 Containment-Target Views Are Enough

We now show that when looking for a minimum-size viewset (either conjunctive or disjunctive) for a conjunctive query workload and database instance, we only need to consider containment-target views, assuming all rewritings are conjunctive. In addition, there is a linear upper bound, in the size of the input query workload, on the number of views in any such viewset.

**Lemma 3.1** Given a database instance, for any conjunctive query workload \( Q \) and for any minimum-size disjunctive viewset \( V \) for \( Q \) (assuming conjunctive rewritings), each view in \( V \) is a containment-target view for at least one query in the workload \( Q \).

**Proof:** In looking for a viewset that is a minimum-size solution for a query workload \( Q \) and database instance \( D \), one way to minimize the total size of the viewset is to remove those views that are not necessary to obtain equivalent rewritings of all the queries in \( Q \). In particular, minimum-size viewsets cannot contain filtering views for any query in \( Q \), as those views are not necessary in producing equivalent query rewritings.

**Theorem 3.2** Given a database instance, for any conjunctive query workload \( Q \) and for any minimum-size disjunctive viewset \( V \) for \( Q \) (assuming conjunctive rewritings), if \( n \) is the total number of subgoals in all the queries in the workload \( Q \), then the viewset \( V \) has at most \( n \) views.

**Proof:**

1. By Lemma 3.1, given a database instance, for any \( Q \) and \( V \) that satisfy the conditions of this theorem, each view in \( V \) is a containment-target view for at least one query in \( Q \).

2. For each query \( q \) in \( Q \), to produce an equivalent rewriting of \( q \) it is necessary to cover all the subgoals of \( q \) using views.
3. In a minimum-size viewset \( \mathcal{V} \), there is no need to have more than one (containment-target) view to cover any subgoal of any query in \( \mathcal{Q} \).

It has been shown [LMSS95] that for any conjunctive query with \( n \) subgoals, if the query has a rewriting using views, then there exists a rewriting with at most \( n \) views. At the same time, that result did not provide any optimality guarantees for the views in the rewriting.

Even though Lemma 3.1 says that in searching for a minimum-size viewset, we can restrict our consideration to containment-target views only, the search space of views for a query workload can still be very large, even if we examine conjunctive views only. There are mainly two reasons: (1) there are many ways to choose subsets of the query subgoals; and (2) there are many ways to project out variables in a view definition. The following example illustrates the point.

**Example 3.2** For an integer value \( k \geq 1 \), consider a query workload \( \{Q_k\} \), where:

\[
Q_k : \text{ans}(X, Y_1, \ldots, Y_k) :- p_1(X, Y_1), \ldots, p_k(X, Y_k).
\]

We can define at least the following containment-target views for \( \{Q_k\} \). Each view is defined as a subset of the subgoals of \( Q_k \), and all variables in each view are distinguished. It is easy to see that by taking conjunctions of some of these views, we can obtain many different rewritings of \( Q_k \). Further, the total number \( N \) of these containment-target views is \( N = 2^k - 1 \). Notice that the set of possible containment-target views for the workload \( \{Q_k\} \) will be even larger if we also consider views with nondistinguished variables.

Based on this example, we make the following observation.

**Observation 1** Given a database instance, for the problem of finding a minimum-size viewset for a workload of conjunctive queries, the size of the search space of views can be (at least) exponential in the length of the definitions of the queries.

### 3.4 Disjunctive Views Can Provide Better Solutions

A query has a self-join if the minimized query definition [CM77] has at least two subgoals with the same relation name. When workload queries have self-joins, we show that it might be better to materialize disjunctive views than conjunctive views.

**Proposition 3.1** There exists a query workload and a database instance, such that a solution with disjunctive views requires strictly less storage space than any solution using conjunctive views only.

**Proof:** We give the proof by constructing such an example. Let \( \text{flight}(\text{source}, \text{destination}) \) be a base table, in which a tuple \( \text{flight}(s, t) \) means that there is a direct flight from city \( s \) to city \( t \). A query workload \( \{Q\} \) has a single query \( Q \) that asks for all sequences of airports that give one-stop flights:

\[
Q : \text{ans}(X, Y, Z) :- \text{flight}(X, Y), \text{flight}(Y, Z).
\]

We define a disjunctive view \( V = V_1 \cup V_2 \), where
Figure 3: A database of the “flight” relation.

\[ V_1: \quad \text{ans}(X, Y) \leftarrow \text{flight}(X, Y), \text{flight}(Y, Z). \]
\[ V_2: \quad \text{ans}(Y, Z) \leftarrow \text{flight}(X, Y), \text{flight}(Y, Z). \]

In the definition of \( V \), \( V_1 \) gives all pairs of airports that are the “first hop” in any one-stop flight, whereas \( V_2 \) provides the pairs of airports for all “second hops” in such flights. Note that the relations for \( V_1 \) and \( V_2 \) will share tuples if there exist flights with at least two stops.

For the database in Figure 3, we show that the disjunctive viewset \( \{ V \} \) is smaller than any conjunctive viewset. Let \( W \) be any optimal conjunctive solution for the workload. By Lemma 3.1, the viewset \( W \) consists of conjunctive containment-target views only. We show that for the database in Figure 3, the total number of occurrences of all data values in the solution \( W \) is greater than the respective measure in the disjunctive solution \( \{ V \} \).

Let \( P \) be an equivalent rewriting of the query \( Q \) using the views in \( W \).

\[ P: \quad \text{ans}(A, B, C) \leftarrow w_1(X_1), \ldots, w_k(X_k). \]

By Theorem 3.2, \( P \) has no more than two view literals, i.e., \( k \leq 2 \). There are two cases: (1) each view \( w_i \) is used once in the rewriting \( P \), or (2) some view \( w_j \) is used more than once in \( P \). We consider the two cases separately.

**Case 1:** Each view \( w_i \) is used exactly once in rewriting \( P \). From Theorem 3 in [CG00], each view in \( P \) can be defined as a subexpression of the query \( Q \). Thus, all views in the rewriting \( P \) come from a set \( S \) of all conjunctive containment-target views that can be defined as subexpressions of the query \( Q \); this set has just four views. By considering all combinations \( W \) of the views in the set \( S \) that produce equivalent rewritings of \( Q \) and by computing the sizes of the resulting viewsets on the database in Figure 3, we obtain that the disjunctive solution \( \{ V \} \) has fewer occurrences of all data values than any such viewset \( W \).

**Case 2:** Some conjunctive view \( w_j \) can be used more than once in the rewriting \( P \). Because \( P \) has at most two view subgoals, the only possibility in this case is that \( P \) is a self-join of a single conjunctive view:

\[ P: \quad \text{ans}(A, B, C) \leftarrow w(X_1), w(X_2). \]

By Theorem 3.1 in [CHS01], in this case views in \( P \) can have more subgoals than the query \( Q \). Suppose there exists a conjunctive view \( w \) that, after minimization, has more subgoals than the query \( Q \). Then the minimized definition of \( w \) has at least three subgoals of the flight relation. In addition, the definition of \( w \) cannot have cross-products; otherwise, the solution \( \{ w \} \) would be suboptimal for \( Q \) and for the database in Figure 3. By considering all possible combinations of nontrivial equality join predicates on three flight subgoals, we verify that the three subgoals in \( w \) cannot contain any subset of subgoals of the query \( Q \). In assuming that a view can be used in the rewriting \( P \) and have more subgoals than \( Q \), we have arrived at a contradiction. On the other hand, suppose the view \( w \) is a
subexpression of the query \( Q \) (i.e., \( w \) is in the set \( S \), see case (1)). Then for any equivalent rewriting of the query \( Q \) that is a self-join of \( w \), the viewset \( \{w\} \), on the given database, has more data values than the disjunctive solution \( \{V\} \).

In summary, for both cases we have shown that on the given instance of relation \( \text{flight} \), an optimal conjunctive viewset requires strictly more storage space than the disjunctive viewset \( \{V\} \). Therefore, the disjunctive viewset \( \{V\} \) is smaller than any conjunctive viewset.

3.5 Disjunctive Views and Rewritings: The Problem Is Decidable

In this section we show that if we allow disjunctive views in viewsets, then the problem of finding an optimal solution for a workload of conjunctive queries is still decidable, even if we also allow disjunctions in query rewritings. Let a minimum-size disjunctive viewset for a workload of conjunctive queries be a minimum-size viewset for the workload in the space of disjunctive views. Any or all of its disjunctive views can be purely conjunctive.

**Theorem 3.3** Given a database instance and a finite workload of conjunctive queries, and assuming conjunctive rewritings only, we can construct a finite search space of views that includes all views in at least one minimum-size disjunctive viewset for the query. The number of views in the search space is at most triply-exponential in the sum of lengths of the definitions of the workload queries. \( \square \)

**Proof:** We show that for any conjunctive query workload \( Q \) and database \( D \), to obtain all nontrivial disjunctive views in at least one minimum-size disjunctive viewset \( \mathcal{U} \), we can consider rewritings of the queries in \( Q \) using sets of representative conjunctive views defined in Note 1 to Theorem 3.1; unions of some of those conjunctive views are the disjunctive views in \( \mathcal{U} \). The proof of the theorem follows from this claim and from the bound given by Theorem 3.1 on the number of such conjunctive views. We prove the claim first for singleton query workloads and then for non-singleton workloads.

**Case 1:** Let the query workload \( Q \) have just one query \( Q \), and let \( \mathcal{V} \) be a minimum-size disjunctive solution for \( Q \). Consider a rewriting of the query \( Q \) using \( \mathcal{V} \), and consider the expansion of the rewriting in terms of the conjunctive components \( \mathcal{V}' \) of the views in \( \mathcal{V} \). The expansion is a union of several conjunctive queries, and one of the queries, \( P \), is equivalent to the query \( Q \) [SY80]. We call \( P \) the equivalent conjunct of \( Q \) in terms of \( \mathcal{V}' \); let \( P^{\exp} \) be the expansion of \( P \) in terms of the schema of the database \( D \).

Suppose the viewset \( \mathcal{V} \) contains exactly one nontrivial disjunctive view, \( V \), that is a union of two conjunctive views, \( V_1 \) and \( V_2 \). Consider an alternative conjunctive solution \( \mathcal{W} \) that is obtained by replacing, in \( \mathcal{V} \), this disjunctive view \( V \) by \( V_1 \) and \( V_2 \). \( \mathcal{W} \) is a solution for the query \( Q \) because of the conjunct \( P \). We say the viewset \( \mathcal{V} \) is a better solution for the query \( Q \) than \( \mathcal{W} \) if \( \text{size}(\mathcal{V}) < \text{size}(\mathcal{W}) \) on the database \( D \). Note that \( \text{size}(\mathcal{V}) \) can be less than \( \text{size}(\mathcal{W}) \) only when the equivalent conjunct \( P \) (of \( Q \) in terms of \( \mathcal{V}' \)) has a join of \( V_1 \) and \( V_2 \). (If \( P \) has either just one conjunct of the disjunctive view \( V \), or if \( P \) has a self-join of just one conjunct of \( V \), we could obtain a conjunctive solution \( \mathcal{W}' \) for the query \( Q \) that is at least as good on the database \( D \) as \( \mathcal{V} \), by replacing, in \( \mathcal{V} \), the view \( V \) by that one conjunct.)

Next, we show that we can construct, from the expansion \( P^{\exp} \) of the equivalent conjunct \( P \) of \( Q \), a disjunctive view \( V' \), such that (1) \( V' \) is used in the same way as \( V \) in rewriting \( Q \), and (2) \( \text{size}(\{V'\}) \leq \text{size}(\{V\}) \) on \( D \). The view \( V' \) is a union of the views \( V_1' \) and \( V_2' \), which are the representatives of the views \( V_1 \) and \( V_2 \) in the space \( S \) of all conjunctive views whose length is at most singly-exponential in the length of the query \( Q \). (See Note 1 to Theorem 3.1.) From the properties of
representative views, the result of replacing $V$ with $V'$ in the viewset $\mathcal{V}$ is at least as good a solution for $\{Q\}$ and $D$ as the viewset $\mathcal{V}$. We can find the view $V'$ by testing all unions of two views in $\mathcal{S}$; thus, we can construct a search space of all disjunctive views that are part of at least one optimal solution for $\{(Q), D\}$.

We obtain $V'$ from $P^{\text{exp}}$ as follows. By Theorem 3.2, the maximal number of view subgoals in $P$ is the number of subgoals in the query $Q$. We take all partitions of the subgoals of $P^{\text{exp}}$ into up to $n$ parts; for each partition that has $k$ elements, we design $k$ conjunctive views whose bodies are the elements of the partition; we consider each resulting conjunctive viewset $\mathcal{T}$ that gives an equivalent rewriting of $Q$, such that $P^{\text{exp}}$ is the expansion of the rewriting. (In particular, this procedure gives us the viewset $\mathcal{W}$. Now, for each $\mathcal{T}$, we construct its representative viewset $\mathcal{T}'$ (see proof of Theorem 3.1), including the representative viewset of $\mathcal{W}$. On the database $D$, $\text{size}(\mathcal{T}') \leq \text{size}(\mathcal{T})$, and the number of subgoals in each view in $\mathcal{T}'$ is at most singly-exponential in the number $n$ of subgoals of the query $Q$.

Recall that, under our assumption, $\text{size}(\mathcal{V}) < \text{size}(\mathcal{W})$. We now construct, from each viewset $\mathcal{T}'$, all disjunctive views $V$ that are each a union of two elements of $\mathcal{T}'$. By construction, from the representative viewset of $\mathcal{W}$ we obtain a disjunctive view $V'$ that is contained in $V$ and can replace $V$ in the rewriting of $Q$ in terms of $V$. Because $\mathcal{V}$ is a minimum-size viewset for $\{(Q), D\}$, the result of replacing $V$ by $V'$ in $\mathcal{V}$ is also a minimum-size viewset for $\{(Q), D\}$.

Observe that each such view $V'$ can be generated by taking a union of two conjunctive views in $\mathcal{S}$, and that the process of generating all such views $V'$ is finite because the number of views in $\mathcal{S}$ is finite. From this observation, we obtain the claim for Case 1 of the theorem. (The proof is extended in a straightforward way to cases where the viewset $\mathcal{V}$ has more than one conjunctive view and where any disjunctive view in $\mathcal{V}$ can have more than two conjuncts.)

**Case 2.** Suppose the query workload $Q$ has at least two queries: $Q = \{Q_1, \ldots, Q_m\}$, $m \geq 2$. In addition to finding all disjunctive views that can be used to rewrite individual queries in the workload $Q$, we now want to account for each nontrivial disjunctive view that can be used to equivalently rewrite more than one query in $Q$. To achieve this goal, all we have to do is to replace, in the reasoning for Case 1 above, $P$ (the equivalent conjunct of the only query $Q$ in the workload in Case 1) by a conjunction $\mathcal{P}$, which we obtain as follows:

- we take an equivalent conjunct $P_i$ of each query $Q_i$ ($i \in \{1, \ldots, m\}$) in the workload $Q$,
- if necessary, we rename the variables in the conjuncts $P_1, \ldots, P_m$, to avoid using any variable name in more than one conjunct,
- finally, we take a conjunction $\mathcal{P}$ of all these conjuncts: the body of $\mathcal{P}$ is $P_1 \& \ldots \& P_m$, and the head of $\mathcal{P}$ comprises all head variables of $P_1, \ldots, P_m$.

By applying the reasoning in Case 1 to the individual conjuncts $P_i$ in $\mathcal{P}$, we show that there exists a rewriting $\mathcal{P}'$ that is equivalent to $\mathcal{P}$ on $D$, whose view relations have the same size on $D$ as the relations for minimum-size conjunctive viewsets for $\mathcal{P}$, and such that that the number of subgoals of each disjunctive view for $\mathcal{P}'$ is at most singly-exponential in the sum of the lengths of the queries in the workload $Q$.

For any minimum-size conjunctive viewset $\mathcal{V}$ for the workload $Q$ and assuming conjunctive rewritings only, $\mathcal{V}$ is also a minimum-size conjunctive viewset for the conjunction $\mathcal{P}$. Using the reasoning in Case 1 above, we can find all representative conjunctive viewsets, $\mathcal{W}'$, that can equivalently rewrite the conjunction $\mathcal{P}$; the complexity bounds are the same as in Case 1. All that remains to be done is to find those viewsets among $\mathcal{W}'$ that can be used to equivalently rewrite all individual queries in the workload $Q$. This observation concludes the proof of Case 2 and the proof of the theorem.
3.6 Disjunctive Rewritings Are Not Needed

We now show that it is not necessary to consider nontrivial disjunctive rewritings of query workloads in terms of minimum-size disjunctive viewsets. Purely conjunctive rewritings are all we need to examine.

**Theorem 3.4** Given a database instance, let \( Q \) be an arbitrary finite workload of conjunctive queries, and let \( V \) be any minimum-size disjunctive viewset, such that \( V \) gives an equivalent disjunctive rewriting of each query in the workload \( Q \). Then for each query in the workload \( Q \), there exists an equivalent conjunctive rewriting of the query in terms of the views in \( V \). \( \square \)

**Proof:** Consider an arbitrary finite workload \( Q \) of conjunctive queries. Let \( V \) be any minimum-size disjunctive viewset, such that \( V \) gives an equivalent disjunctive rewriting of each query in the workload \( Q \). For each query \( Q \) in \( Q \), consider an equivalent disjunctive rewriting \( P \) of \( Q \) in terms of the views \( V \). The query \( P \) is a union of conjunctive queries \( P_1, \ldots, P_n \), where \( P_i(\bar{X}) := V_{i1}(\bar{X}_{i1}), \ldots, V_{im_i}(\bar{X}_{im_i}), i \in \{1, \ldots, n\} \). Here, each view \( V_{ij} \) belongs to the viewset \( V \).

By definition, the conjunctive query \( Q \) is equivalent to the union of the expansions of these queries \( P_i, i \in \{1, \ldots, n\} \). Each expansion is a union of conjunctive queries, because each view \( V_{ij} \) may be a disjunctive view. From [SY80], the query \( Q \) is equivalent to one of the expanded queries, which we denote by \( E_k \), and the remaining expanded queries are contained in \( Q \). Let \( E_k \) come from the query \( P_i \) in \( P_1, \ldots, P_n \). Notice that \( P_i \) is an equivalent conjunctive rewriting of \( Q \) using \( V \). We modify \( V \) into \( V' \) by removing all views that do not contribute to the equivalent expanded query for any query in \( Q \). The resulting viewset \( V' \) provides an equivalent conjunctive rewriting of each query in \( Q \); in addition, \( size(V') \leq size(V) \).

**Corollary 3.2** For any finite workload \( Q \) of conjunctive queries and a database instance \( D \), assuming disjunctive rewritings, the problem of finding a minimum-size disjunctive viewset for \( Q \) on \( D \) is decidable. \( \square \)

4 Conjunctive Queries without Self-Joins: The Problem Is in NP

In this section we study workloads of conjunctive queries without self-joins. Figure 4 shows the view spaces we consider to find a minimum-size set of disjunctive views. We first show that if the workload queries do not have self-joins, then we need to consider only conjunctive views, because nontrivial disjunctive views do not add any new solutions (Section 4.1). We then further restrict our consideration to subexpression-type views (Section 4.2), and then to full-reducer views (Section 4.3). We show that the problem of finding an optimal disjunctive viewset for queries without self-joins is in NP, precisely because we can always find a minimum-size viewset in the full space of disjunctive views by searching just a very restricted search space of full-reducer views.

4.1 Conjunctive Views Are Enough

**Theorem 4.1** Suppose a set \( V \) of disjunctive views is a solution for a given database instance \( D \) and workload \( Q \) of conjunctive queries without self-joins. Then there exists another solution \( V' \) for \( D \) and \( Q \), such that all views in \( V' \) are conjunctive, and \( size(V') \leq size(V) \). \( \square \)
**Proof:** We first look at singleton query workloads and then extend our observations to arbitrary query workloads.

**Case 1** (singleton query workloads only). Let \( Q \) be any singleton query workload, \( Q = \{Q\} \), where the query

\[
Q : \text{ans}(\tilde{Z}) \leftarrow R_1(\tilde{Y}_1), \ldots, R_n(\tilde{Y}_n).
\]

is conjunctive and does not have self-joins. Let \( D \) be an arbitrary database instance, and let a set \( V \) of disjunctive views be a solution for \( Q \) and \( D \). Then there exists an equivalent rewriting \( P \) of \( Q \) using the views in \( V \). Without loss of generality, we assume that all views in \( V \) are used in \( P \). Consider each nontrivial disjunctive view

\[ V = V_1 \cup V_2 \cup \cdots \cup V_k, \]

\( k > 1 \), that is used in \( P \); \( V_1, \ldots, V_k \) are conjunctive views. The idea of the proof is that there exists a transformation \( \mu \) of the view \( V \) that produces a new disjunctive view:

\[ V' = V'_1 \cup V'_2 \cup \cdots \cup V'_k, \]

where \( V'_i = \mu(V_i) \), \( i = 1, \ldots, k \). The new disjunctive view \( V' \) is used in a new equivalent rewriting \( P' \) of the query \( Q \). We show that for any conjunctive component \( V'_i \) of the view \( V' \), \( V'_i \) can replace the entire view \( V \) in the equivalent rewriting \( P' \). Therefore, we can replace the disjunctive view \( V \) with a conjunctive view \( V'_i \), where \( |V'_i| \leq |V'| \leq |V| \).

Now we give the details of the proof. Assume there are \( m \) occurrences of \( V \) in \( P \):

\[ P : \text{ans}(\tilde{X}) \leftarrow V(\tilde{X}_1), \ldots, V(\tilde{X}_m), G. \]

where each of \( \tilde{X}, \tilde{X}_1, \ldots, \tilde{X}_m \) is a list of arguments, and \( G \) represents the instances of other views that are not \( V \). By replacing all the disjunctive views by their definitions as unions of conjuncts, we get a union of conjunctive queries, which is equivalent to the query \( Q \). From [SY80], at least one of these conjunctive queries — that we denote it by \( P' = P_{\text{exp}} \equiv Q \), where \( P_{\text{exp}} \) is an expansion of \( P \) [CM77]. Let \( \mu \) be a containment mapping from \( P_{\text{exp}} \) to \( Q \). By applying \( \mu \) on \( P \), we get another equivalent rewriting:

\[ P' : \text{ans}(\tilde{X}') \leftarrow V(\tilde{X}_1'), \ldots, V(\tilde{X}_m'), G'. \]

where \( \tilde{X}' = \mu(\tilde{X}), \tilde{X}_i' = \mu(\tilde{X}_i), i = 1, \ldots, m \). \( G' \) represents the subgoals in \( P' \) that do not use the conjunctive components in \( V \). In the definition of \( V \), \( V = V_1 \cup V_2 \cup \cdots \cup V_k \), we remove all the \( V_i \)'s that do not appear in \( P' \). Without loss of generality, \( P' = \mu(P) \) can be represented as:

\[ P' : \text{ans}(\tilde{X}') \leftarrow V_1(\tilde{X}_1'), \ldots, V_m(\tilde{X}_m'), G'. \]

where each \( \tilde{V}_i \) is a conjunctive component of the view \( V \).
Consider the conjunctive view $\hat{V}_1$ and its corresponding contained rewriting (in $\mathcal{P}'$) that does not use the conjunctive components $\hat{V}_2, \ldots, \hat{V}_m$ of $V$:

$$H : \text{ans}(\hat{X}') \vdash \hat{V}_1(\hat{X}'_1), \ldots, \hat{V}_1(\hat{X}'_m), G.$$ 

Let $\beta$ be a containment mapping from $Q$ to $H^{\text{exp}}$. For any $j \geq 2$, we show that for all the local mappings (of $\beta$) from $Q$ to the subgoals/arguments in $\hat{V}_1(\hat{X}'_j)^{\text{exp}}$, we can “redirect” them to the corresponding subgoals/arguments in the expansion of $\hat{V}_1(\hat{X}'_j)$ and thus get another containment mapping $\beta'$ from $Q$ to $H^{\text{exp}}$, where the images of the subgoals of $Q$ do not come from the expansion of any $\hat{V}_1(\hat{X}'_j)^{\text{exp}}$, $j \geq 2$.

\[
\begin{align*}
Q : \text{ans}(Z) & : \ldots R(\ldots W \ldots) \\
\beta & \mapsto \beta' \\
H^{\text{exp}} : \text{ans}(Z) & : \ldots R(\ldots Y \ldots) \ldots R(\ldots Y \ldots) \\
H : \text{ans}(Z) & : \hat{V}_1(\hat{X}'_1), \ldots, \hat{V}_1(\hat{X}'_j), \ldots, \hat{V}_1(\hat{X}'_m), G
\end{align*}
\]

Figure 5: Redirecting the mapping $\beta$ to $\beta'$.

Here are the details on redirecting the mappings. Consider each subgoal $R(\ldots Y \ldots)$ in $\hat{V}_1(\hat{X}'_j)^{\text{exp}}$, where $j \geq 2$. Let $R(\ldots W \ldots)$ be the corresponding query subgoal in $Q$. Because $Q$ does not have self-joins, it has only one instance of relation $R$. Here both $Y$ and $W$ are the $l$-th argument for relation $R$. There are two possible cases.

1. $Y$ is a distinguished variable of $\hat{V}_1$. That is, in the definition of $\hat{V}_1$, there is at least one $R$-subgoal whose $l$-th attribute appears in the head of $\hat{V}_1$. Since $R(\ldots W \ldots)$ is the only $R$-subgoal in $Q$, the mapping $\mu$ guarantees that $Y = W$. In addition, the expansion of $\hat{V}_1(\hat{X}'_j)$ has a subgoal $R(\ldots W \ldots)$ because of the mapping $\mu$. We then redirect the mapping from $W$ in $Q$ — which used to be to $W$ in the expansion of $\hat{V}_1(\hat{X}'_j)$ — to $W$ in the expansion of $\hat{V}_1(\hat{X}'_j)$.

2. $Y$ is a nondistinguished variable of $\hat{V}_1$, i.e., $Y$ is a fresh variable in the expansion of $\hat{V}_1(\hat{X}'_j)$. Then $W$ must also be a nondistinguished variable of $Q$. The expansion of $\hat{V}_1(\hat{X}'_j)$ also provides a corresponding fresh variable $Y'$ that can be used as the image of $W$. Notice that in this case, $\hat{V}_1(\hat{X}'_j)^{\text{exp}}$ and $\hat{V}_1(\hat{X}'_j)^{\text{exp}}$ provide all instances of $Y'$ and $Y$, respectively [PL00, ALU01]. So we can redirect the mappings — which used to be to $Y$ in $\hat{V}_1(\hat{X}'_j)^{\text{exp}}$, $j \geq 2$ — to $Y'$ in $\hat{V}_1(\hat{X}'_j)^{\text{exp}}$.

By redirecting all the local mappings from the expansion of $\hat{V}_1(\hat{X}'_j)$, $j \geq 2$, to the subgoals in the expansion of $\hat{V}_1(\hat{X}'_j)$, we have obtained, from the containment mapping $\beta$, another containment mapping $\beta'$, from $Q$ to the expansion of the following rewriting:

$$H_{\text{new}} : \text{ans}(\hat{X}') \vdash \hat{V}_1(\hat{X}'_j), G.$$ 

The containment mapping $\beta'$ implies that $H_{\text{new}}^{\text{exp}} \sqsubseteq Q$. Since $\mathcal{P}'$ is an equivalent rewriting of $Q$, we obtain that $Q$ is contained in the expansion of $\mathcal{P}'$. In addition, the expansion of $\mathcal{P}'$ is also contained

---

1Note that we cannot do the redirection in this manner if the relation $R$ appears more than once in $Q$; see the flight example in Section 3.4.
in \( H_{\text{new}}^{\text{exp}} \), since the subgoals in \( H_{\text{new}} \) are a subset of those in \( \mathcal{P}' \). Thus \( Q \subseteq H_{\text{new}}^{\text{exp}} \). So \( H_{\text{new}} \) is also an equivalent rewriting of \( Q \). Notice that in \( H_{\text{new}} \),  of all the conjunctive components of \( V \) we use only one component \( \hat{V}_i \). Thus we can replace the disjunctive view \( V \) with a conjunctive view \( \hat{V}_i \).

By doing this replacement for all the disjunctive views in \( \mathcal{V} \), we get a set \( \mathcal{V}' \) of purely conjunctive views that can answer the query \( Q \). By construction, \( \mathcal{V}' \) does not require more storage space than \( \mathcal{V} \).

**Case 2** (arbitrary query workloads). Let \( Q \) be a query workload, such that \( Q \) has at least two conjunctive queries, \( Q = \{Q_1, \ldots, Q_n\} \), \( n \geq 2 \), and such that all queries in \( Q \) are queries without self-joins. Let \( \mathcal{V} \) be a disjunctive viewset that is a solution for \( Q \) and for an arbitrary database instance \( \mathcal{D} \). If, for any nontrivial disjunctive view \( V \) in \( \mathcal{V} \), there exists at most one query in \( Q \), such that the rewriting of the query (in terms of \( \mathcal{V} \)) uses the view \( V \), then this case reduces to Case 1 above. In the remainder of the proof we assume that in the viewset \( \mathcal{V} \), there is a nontrivial disjunctive view \( V = V_1 \cup \ldots \cup V_n \) (each \( V_i \) is purely conjunctive), such that \( V \) is used to rewrite at least two queries in the workload \( Q \).

Without loss of generality, let \( Q_1 \) and \( Q_2 \) be two such queries. Using the reasoning in Case 1 above, we find that there are two conjuncts, \( V_1 \) and \( V_2 \), of the view \( V \), such that the query \( Q_1 \) (resp. \( Q_2 \)) has a conjunctive rewriting that uses just the conjunct \( V_1 \) (resp. \( V_2 \)) of \( V \). (We assume here that \( V \) is the only nontrivial disjunctive view used in the rewritings of the queries \( Q_1 \) and \( Q_2 \); it is straightforward to generalize the proof to the case of multiple disjunctive views used in the rewritings of multiple queries in the workload \( Q \).) In the remainder of the proof, we show that \( V_1 \equiv V_2 \). It follows that if a nontrivial disjunctive view \( V \) is used to rewrite more than one query in the workload \( Q \), then there exists a conjunct \( \hat{V} \) of the view \( V \), such that \( \hat{V} \) can be used to construct equivalent conjunctive rewritings of all those queries. This observation concludes our proof.

We now show that \( V_1 \equiv V_2 \). Using the reasoning in Case 1, we can show that the conjunct \( V_1 \) contains the self-join \( V_2 \times \ldots \times V_2 \) in the expansion of the rewriting of the query \( Q_1 \) (this rewriting uses the view \( V \)). Similarly, we can observe that the conjunct \( V_2 \) contains the self-join \( V_1 \times \ldots \times V_1 \) in the expansion of the rewriting of the query \( Q_2 \). Using these self-joins and using the reasoning in Case 1, we construct two redirected containment mappings: (1) a mapping \( \mu_1 \) from \( V_1 \) to just one occurrence of \( V_2 \) in the self-join \( V_2 \times \ldots \times V_2 \), and (2) a mapping \( \mu_2 \) from \( V_2 \) to just one occurrence of \( V_1 \) in the self-join \( V_1 \times \ldots \times V_1 \). Because both \( V_1 \) and \( V_2 \) are conjunctive views, we conclude that in the disjunctive view \( V \), the conjuncts \( V_1 \) and \( V_2 \) are equivalent. This observation concludes the proof of the theorem.

\[
\square
\]

### 4.2 Subexpression-Type Views Are Enough

We saw in Section 3 that to find a minimum-size conjunctive viewset for a workload of conjunctive queries, it is enough to consider views whose definition is at most exponential in the length of the longest query definition in the workload. Now we show that for workloads of queries without self-joins, we can further reduce the size of the search space of views by considering only those conjunctive views that are defined as subexpressions of the workload queries. By considering such views only, we can still find a conjunctive viewset that is an optimal solution for the given problem input in the space of disjunctive views. We begin by showing the following result.

**Theorem 4.2** Given a database instance, for any conjunctive query \( Q \) without self-joins there is a minimum-size conjunctive viewset \( \mathcal{U} \), such that each view in \( \mathcal{U} \) is a subexpression of \( Q \), with possibly attributes projected. \( \square \)
Proof: We prove the theorem by using a modification of the proof of Theorem 3 in [CG00]. Suppose a viewset \( \mathcal{V} \) is some (not necessarily minimum-size) conjunctive viewset, such that the query \( Q \) has a rewriting using \( \mathcal{V} \). From the viewset \( \mathcal{V} \) we construct another viewset, \( \mathcal{W} \), such that \( \mathcal{W} \) also provides a rewriting of the query \( Q \) and has the following properties. The amount of space required to store \( \mathcal{W} \) does not exceed the space required to store \( \mathcal{V} \), on any database, and each view in \( \mathcal{W} \) is defined as a subexpression of the query \( Q \), with possibly attributes projected.  

Theorems 4.1 and 4.2 and the results in Section 3 imply that to find a minimum-size disjunctive viewset for a workload of conjunctive queries without self-joins, it is enough to consider conjunctive containment-target views whose body is a subexpression of at least one query in the workload. As a result, we have reduced the size of the search space of views that includes all views in all minimum-size disjunctive viewsets, from triply-exponential to singly-exponential in the size of the query workload.

**Corollary 4.1** For any database instance and any single conjunctive query without self-joins, we can construct a finite search space of views that includes all views in at least one minimum-size disjunctive viewset for the query, such that the number of views in the search space is at most exponential in the size of the query definition.  

4.3 Full-Reducer Views Are Enough

Now we further reduce the size of the search space of views for a single conjunctive query without self-joins, by considering only full-reducer views. A full-reducer view is a view whose body is the query body [Yan81]. Consider Example 3.2 again. In that example, the body of each view can be replaced by the full body of the query \( Q_k \). After the replacement, the number of tuples in each view cannot increase. More precisely, none of the views will have any dangling tuples after the replacement. At the same time, for each such view, there exists a database where the view is part of some minimum-size viewset for the query \( Q_k \). We formalize these observations in the following result.

**Lemma 4.1** Given a database instance, consider any conjunctive query \( Q \) without self-joins; let \( \mathcal{V} \) be any solution for \( Q \). For each view \( V \in \mathcal{V} \) that is not a full-reducer view, we can construct from the view \( V \) a new view \( W \), by replacing the body of \( V \) with the body of \( Q \). Then the resulting viewset \( \mathcal{V}' \) is also a solution for \( Q \), and \( \text{size}(\mathcal{V}') \leq \text{size}(\mathcal{V}) \).  

**Proof:** For any conjunctive query \( Q \) without self-joins and for its conjunctive solution \( \mathcal{V} \), consider any containment-target view \( V \in \mathcal{V} \), such that \( V \) is defined as a proper subexpression of the query. From each such view \( V \), we can construct a full-reducer view \( W \), by simply adding to the definition of \( V \) the missing subgoals of the query \( Q \). We can show that \( W \) is contained in \( V \). Therefore, on any database we have \( \text{size}(W) \leq \text{size}(V) \). Now consider an equivalent rewriting \( P \) of \( Q \) in terms of the viewset \( \mathcal{V} \). If, in the rewriting \( P \), we replace \( V \) by \( W \), the resulting rewriting \( P' \) will still be equivalent to \( Q \), as the containment mappings between \( Q \) and the expansion of \( P \) can be extended to mappings between \( Q \) and the expansion of \( P' \).  

**Theorem 4.3** Given a database instance, for any conjunctive query without self-joins, there exists a minimum-size disjunctive viewset \( \mathcal{V} \), such that each view in \( \mathcal{V} \) is a conjunctive full-reducer containment-target view.  

We can extend the result of Theorem 4.3 to queries defined using comparisons with constants that are not equality comparisons.
Corollary 4.2  Given a database instance, for any conjunctive query without self-joins that may have comparisons with constants of arbitrary types, there exists a minimum-size disjunctive viewset \( \mathcal{V} \), such that each view in \( \mathcal{V} \) is a conjunctive full-reducer containment-target view.

Proof: Consider an arbitrary conjunctive query \( Q \) without self-joins, such that \( Q \) is defined on a database schema \( S \). We show how to construct for \( Q \) an equivalent rewriting in terms of a minimum-size disjunctive viewset \( \mathcal{V} \), such that each view in \( \mathcal{V} \) is a conjunctive full-reducer containment-target view defined on the schema \( S \).

For the proof, we use the following construction. We define a new database schema \( S_{\text{comp}} \), as follows: For each literal \( p(A_1, \ldots, A_n) \) in the body of \( Q \), such that \( C(p) \) are all the comparisons of arguments of \( p \) with constants in the query \( Q \), we define a new relation \( p'(A_1, \ldots, A_n) : - p(A_1, \ldots, A_n), C(p) \). For any database \( D \) with schema \( S \) and its associated database \( D_{\text{comp}} \), the relations \( Q(D) \) and \( Q_{\text{comp}}(S_{\text{comp}}) \) are equal as sets, as evidenced by the transformation rule "push selections as far as they go," which is used in standard System-R-style query optimizers for relational database systems [SAC+79].

1. The query \( Q \) can be rewritten as a query \( Q_{\text{comp}} \equiv Q \), using the definitions of the relations in the schema \( S_{\text{comp}} \); the rewriting is a straightforward use of the definitions in \( S_{\text{comp}} \). Indeed, for any database \( D \) with schema \( S \) and its associated database \( D_{\text{comp}} \), the relations \( Q(D) \) and \( Q_{\text{comp}}(S_{\text{comp}}) \) are equal as sets, as evidenced by the transformation rule "push selections as far as they go," which is used in standard System-R-style query optimizers for relational database systems [SAC+79].

2. By construction, the query \( Q_{\text{comp}} \) does not have comparisons with constants. Thus, by Theorem 4.3, for the query \( Q_{\text{comp}} \) there exists an equivalent rewriting, \( R_{\text{comp}} \equiv Q_{\text{comp}} \), in terms of a minimum-size disjunctive viewset \( \mathcal{V}_{\text{comp}} \), such that each view in \( \mathcal{V}_{\text{comp}} \) is a conjunctive full-reducer containment-target view. Note that each view in \( \mathcal{V}_{\text{comp}} \) is defined on the database schema \( S_{\text{comp}} \) and has the body that is the same as the body of the query \( Q_{\text{comp}} \).

3. We now show that from the rewriting \( R_{\text{comp}} \) we can produce a rewriting \( R \equiv R_{\text{comp}} \) in terms of views defined on the schema \( S \), such that \( R \) is equivalent to \( R_{\text{comp}} \). Indeed, we can obtain a new viewset \( \mathcal{V} \), by equivalently rewriting each view in the viewset \( \mathcal{V}_{\text{comp}} \); the rewriting substitutes the body of the query \( Q \) for the body of the view. By construction, each view in \( \mathcal{V} \) is a full-reducer view for \( Q \).

From these observations and from transitivity of equivalence, it follows that \( R \) is an equivalent rewriting of \( Q \), \( R \equiv Q \). To conclude the proof, it is enough to observe that for any database \( D \), the size of the viewset \( \mathcal{V} \) is the same as the size of the viewset \( \mathcal{V}_{\text{comp}} \) on the associated database \( D_{\text{comp}} \), and to recall that \( \mathcal{V}_{\text{comp}} \) is a minimum-size viewset for \( Q_{\text{comp}} \equiv Q \).

We now observe that to find an optimal solution for a conjunctive query, it is enough to consider those full-reducer views whose head projects only "necessary" body attributes, that is, just head attributes of the rewriting and the attributes required for joins with the remaining views. Formally, let \( G \) be an arbitrary subset of the relations in the query. Let \( G' \) be the remaining relations in \( Q \). We say an attribute \( A \) of a relation in \( G \) is a relevant attribute of \( G \) if either (1) \( A \) is an output (i.e., distinguished) attribute of \( Q \), or (2) \( A \) is a join attribute in the query, i.e., in \( Q \), this attribute appears

---

2 Possible comparison types include inequality comparisons and string comparisons, such as LIKE in SQL.
in a subgoal of a relation in $G$ and a subgoal of a relation in $G'$. A full-reducer view $V_G$ is called the 
associated view of $G$ if the head arguments $\Sigma_G$ of $V_G$ are exactly the relevant attributes of $G$.

For example, the view $V_1$ in Figure 2 is associated with the subset $G_1 = \{\text{customer, order}\}$. The relevant attributes of the view $V_1$ include output attributes of the query $Q_1$ (Figure 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 rewriting of the query $Q_1$.

<table>
<thead>
<tr>
<th>Relation subset</th>
<th>Associated view</th>
</tr>
</thead>
<tbody>
<tr>
<td>{customer}</td>
<td>$W_1$</td>
</tr>
<tr>
<td>{order}</td>
<td>$W_2$</td>
</tr>
<tr>
<td>{lineitem}</td>
<td>$W_3 = V_2$</td>
</tr>
<tr>
<td>{customer, order}</td>
<td>$W_4 = V_1$</td>
</tr>
<tr>
<td>{customer, lineitem}</td>
<td>$W_5$</td>
</tr>
<tr>
<td>{order, lineitem}</td>
<td>$W_6$</td>
</tr>
<tr>
<td>{customer, order, lineitem}</td>
<td>$W_7 = Q_1$</td>
</tr>
</tbody>
</table>

4.4 Workloads of Queries without Self-Joins: The Problem Is in NP

Notice that, from the update to Example 3.2 that we suggested in Section 4.3, it follows that the search space of conjunctive full-reducer containment-target views can still be exponential in the length of the query definition. At the same time, we have the following powerful result.

**Theorem 4.4** Given a database instance, for any finite workload of conjunctive queries without self-joins, the problem of finding a minimum-size disjunctive viewset is in NP.

**Proof:** Consider a query $Q$ with $n$ subgoals, a database $\mathcal{D}$, and an integer $K$. To check whether a conjunctive viewset $\mathcal{V}$ is a solution for the query $Q$, such that storing the views in the database $\mathcal{D}$ will require at most $K$ bytes, we can do two things. First, to see whether the viewset $\mathcal{V}$ gives an equivalent rewriting of query $Q$, we need to check a witness that provides (1) a rewriting of the query in terms of the views, and (2) the containment mappings between the query and the rewriting. Second, to check whether storing the views in the database $\mathcal{D}$ will require at most $K$ bytes, it is enough to add up the sizes of the views in $\mathcal{V}$ on the database $\mathcal{D}$. From the results in Sections 3 and 4 it follows that the sizes of the structures we need to examine (and, therefore, the time required to examine them) are polynomial in the size of the query $Q$.

5 Computing a Solution Efficiently

So far we have studied the problem of constructing search spaces of views that contain a minimum-size solution for a query workload. Now we study how to compute such a minimum-size solution efficiently. The efficiency is especially important for cases where we need to compute the solution online. For instance, in a client-server environment, a client issues queries to the server that manages the stored data. The server can decompose the query into views and send the view results to the client to
compute the final answers; in this environment it is critical to compute optimal solutions on the server efficiently.

In the remainder of the paper, we focus on the case of a single conjunctive query without self-joins, whose general definition is

\[ Q : \text{ans} (\bar{X}) \ni R_1 (\bar{X}_1), \ldots, R_n (\bar{X}_n). \]

### 5.1 Overview of the Approach

There are at least two challenges in computing an optimal plan in the space of associated views (Section 4.3): (1) we need to know the size of each view, and (2) we need to compute the results of each view. Consider the first challenge. 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.

The main idea of our approach is the following. Before searching for an optimal plan for a query \( Q \), we modify the query as follows. To the output attributes of \( Q \), we add all its join attributes that are not among its output attributes, and denote the new query \( \hat{Q} \). 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 its output its join attributes \( c_{\text{custkey}}, o_{\text{custkey}}, \) and \( o_{\text{orderkey}} \) in the new query \( \hat{Q}_1 \), the new variable “CK” is used for the join attribute “custkey”:

\[
\hat{Q}_1 (N, CK, OD, SP, C, OK, QT, SM) := \text{customer}(CK, N', BLDC'), \]
\[
\text{orders}(OK, CK, OD, SP, C), \]
\[
\text{lineitem}(LN, OK, QT, SD, SM).
\]

We execute the query \( \hat{Q} \), and use its results \( D_{\hat{Q}} \) to estimate the size of each view by doing sampling. To estimate the number of distinct tuples in a view, we treat all values, taken together, in a tuple as a single value. 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, the Smoothed Jackknife Estimator and Shlosser’s Estimator [HNSS95]. The resulting estimates are likely to be more accurate than those obtained using standard size-estimation functions and database statistics. The results of the views can also be computed by either running the views or doing projections on the results of this new query.

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.

Given the estimated size of each view, now we study how to find an optimal decomposition plan for a query \( Q \) using the associated views.

### 5.2 Basic Version of the Algorithm

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 \( G_i \).
Figure 6 shows the basic version of the search algorithm. In the first step, the algorithm estimates the size of each view. All view 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
- $D_{\tilde{Q}}$: answer to the extended query $\tilde{Q}$ 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 $D_{\tilde{Q}}$;
      add $(V_G, S_G)$ to table $S$;
   }
Step 2: // find an optimal decomposition plan
   minValue = $\infty$; 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 < minValue) {
         minValue = size;
         optPlan = plan;
      }
   }
   return (optPlan, minValue);

Figure 6: Basic version of the algorithm.

### 5.3 Pruning Rules

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. We propose pruning rules to reduce the size of the search space while still allowing us to find an optimal plan. Before presenting the rules, we review the concept of "join graphs" [Fin82, Zlo77] 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 $R_i.A = R_j.B$, 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 $R_j.A$.

To illustrate the use of our pruning rules, we use the following simplified schemas of the TPC-H relations. 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)
```
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 s.nationkey = n.nationkey
    AND n.regionkey = r.regionkey
    AND r.name = 'AMERICA';

Figure 7: Query $Q_2$

\[
\begin{align*}
nation(nationkey, name, regionkey) \\
region(regionkey, name)
\end{align*}
\]

Consider the query $Q_2$ in Figure 7; $Q_2$ is a slight variation on Query 2 in the TPC-H benchmark. The join graph of the query $Q_2$ is shown in Figure 8. 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 8: Join graph $G(Q_2)$ of query $Q_2$.

We first extend $Q_2$ by adding to the output attributes its join attributes that are not output attributes, such as $p.partkey$, $n.nationkey$, etc. We execute the extended query $Q_2$ on the server, and get the results $D_{Q_2}$. 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.

5.3.1 Pruning 1: Ignoring Plans that Use “Disconnected” Views

Using this pruning rule, we never consider one class of views, which we call “disconnected views.” As discussed in Section 4.3, given a query, we associate one view with each subset $G$ of the relations. Disconnected 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 disconnected 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 disconnected 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.

5.3.2 Pruning 2: Removing Ear Relations without Output Attributes

Let us look at the \texttt{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, \texttt{nation}. The only contribution of \texttt{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 \texttt{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 \textit{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 [Gra79]. If an ear relation does not have output attributes, 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 one other remaining relation. We repeat this process until we cannot eliminate any more ear relations that do not have output attributes. Note that we may not be able to remove those relations that do not have any output attributes but are connected to more than one other relation, since those relations could be “connecting” two other relations.

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

6 Experiments

We have conducted experiments to evaluate the efficiency and effectiveness of the proposed techniques. We consider client-server environments, where a client issues a query to a server. We want to compute a query-decomposition plan efficiently to reduce the communication cost. We evaluate how efficiently our algorithm can find an optimal decomposition plan and how much communication it can reduce.

6.1 Experimental 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 [TPC]. 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 some of the queries have considerable redundancies. We will see in this section that our approach is effective in removing the redundancies. For our experiments, we removed all the grouping and aggregation operations from the queries. We kept conditions on attributes, such as \texttt{orderdate >= 1997-06-01 and type LIKE 'STANDARD/'}}, since our techniques are applicable in the presence of these conditions (see Corollary 4.2 in Section 4.3). 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 19 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. 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 $q_4$. We used Microsoft SQL Server Version 8.00.294. The program is implemented in C++; it 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 Athalon XP 1.47GHz CPU, 256MB memory, and a 40GB hard disk.

6.2 Estimating View Sizes

We first report the results on the accuracy of our view-size estimators. We have implemented both the Smoothed Jackknife Estimator and the Shlosser's Estimator [HNSS95]. 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 in 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 9 shows how the accuracy changed for different views for queries $q_3$ and $q_4$. The formula for accuracy is $1 - \left\vert \frac{\text{size}_{\text{estimate}} - \text{size}_{\text{real}}}{\text{size}_{\text{real}}} \right\vert$. The experiments corroborated good accuracy of the estimators. In most cases, accuracy was close to 90%.

![Figure 9: Accuracy of view-size estimates on varying data sizes.](image)

6.3 Pruning: The Number of Plans

Table 3 shows the results of applying the pruning rules in Section 5.3 to queries $q_1$ through $q_4$. The experiments show that using the rules can result in a significant reduction in the size of the search space of views.

6.4 Reduction of Data-Communication Costs

We evaluated how much the algorithm in Section 5 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
Table 3: Effect of pruning rules.

<table>
<thead>
<tr>
<th>Queries</th>
<th>Original number of plans</th>
<th>After pruning rule 1</th>
<th>After pruning rule 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$</td>
<td>203</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>$q_2$</td>
<td>15</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>$q_3$</td>
<td>15</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>$q_4$</td>
<td>15</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

reduction ratio for an approach $A$, as the ratio of data size of the 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, as well as other approaches that we will discuss later, such as the compression approach.

Table 4 shows the results on a 60MB database (the scaling factor of 0.06); it can be seen that our technique dramatically reduces the data-communication costs for these queries. For instance, the original size of the answer to the query $q_2$ (the size of NAIVE) was 11.4 MB; by 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 $11.4/1.94 = 5.88$.

Table 4: Data-communication size.

<table>
<thead>
<tr>
<th>Queries</th>
<th>NAIVE (MB)</th>
<th>DECOMP (MB)</th>
<th>Reduction ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$</td>
<td>3.45</td>
<td>2.53</td>
<td>1.36</td>
</tr>
<tr>
<td>$q_2$</td>
<td>11.4</td>
<td>1.94</td>
<td>5.88</td>
</tr>
<tr>
<td>$q_3$</td>
<td>14.3</td>
<td>4.26</td>
<td>3.36</td>
</tr>
<tr>
<td>$q_4$</td>
<td>13.7</td>
<td>5.19</td>
<td>2.64</td>
</tr>
</tbody>
</table>

Table 5 shows the ratio of data-communication reduction 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 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 section, we show that our approach reduces the query-answering time. We consider both 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 sort the view results based on the join attributes whenever possible. (This observation was called “interesting order” in System R [SAC+79].) In computing the final results, we might still need to do some sorting if the needed order of the join attributes for the merge-sort join is different from the available order. In this way, we reduce the time of the client-site join dramatically compared to the transmission time.

### 6.5.1 Different Running Times

Figure 10 shows an example of 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 transferred via a slow network.

![Figure 10: Runtime of $q_2$ on databases of varying data sizes.](image)

Now we compare our DECOMP approach with the NAIVE approach. Our experiments show 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
6.5.2 Total Running Time

Figure 11: Runtime of DECOMP and NAIVE.

Figure 11 compares the running time of DECOMP and NAIVE on queries $q_2$ and $q_3$, on a 30 MB database with the network 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 bar for the NAIVE approach has 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 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 12 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 in each case.

Figure 12: Total running time of NAIVE and DECOMP for varying data sizes.
6.5.3 Running Time and Network Bandwidth

Figure 13 illustrates the relationship between the network bandwidth and 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 uses the logarithmic scale.) Here we show the result for queries $q_2$ and $q_3$ only. From the figure, we can see that our DECOMP approach results in significant time reductions under low network bandwidth. In particular, our approach behaves much better than the 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.

![Running time of NAIVE and DECOMP for varying bandwidths.](image)

6.6 Comparison with Data Compression

Other approaches can be used to minimize the communication cost for transferring query results by reducing the data; data compression is one possibility [CS00]. Our approach is orthogonal to the compression approach; the two approaches could be combined to achieve a better improvement. Here we report our experimental results to verify this claim.

In the 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. Thus, the approach includes four steps: executing the query, compressing the result, transmitting the result, and decompressing the data. Clearly we can combine our approach with compression. For instance, we can first decompose queries into subqueries using our DECOMP approach, and then 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, combining the two approaches results in a better time reduction in transmitting query results.

We now report the results of the experiments that we conducted to verify our analysis. As a compression tool, we chose WinZip [Win] 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 6 lists the reduction ratio for the three approaches on different queries with fixed database size of 60MB. It shows that by combining DECOMP and ZIP, 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 6: Reduction ratio for different queries.

<table>
<thead>
<tr>
<th>Queries</th>
<th>Reduction ratio (DECOMP)</th>
<th>Reduction ratio (ZIP)</th>
<th>Reduction ratio (DECOMP&amp;ZIP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>q_1</td>
<td>1.36</td>
<td>4.3</td>
<td>3.46</td>
</tr>
<tr>
<td>q_2</td>
<td>5.88</td>
<td>7.08</td>
<td>14.16</td>
</tr>
<tr>
<td>q_3</td>
<td>3.36</td>
<td>3.4</td>
<td>12.12</td>
</tr>
<tr>
<td>q_4</td>
<td>2.64</td>
<td>4.27</td>
<td>9.51</td>
</tr>
</tbody>
</table>

Table 7 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 than DECOMP or ZIP in isolation. For instance, for the 60MB database, the ratios of DECOMP, ZIP, and DECOMP&ZIP were 3.36, 3.4, and 12.12, respectively.

Table 7: Reduction ratio of q_3 on varying data sizes.

<table>
<thead>
<tr>
<th>DB size (MB)</th>
<th>Reduction ratio (DECOMP)</th>
<th>Reduction ratio (ZIP)</th>
<th>Reduction ratio (DECOMP&amp;ZIP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2.71</td>
<td>3.49</td>
<td>9.96</td>
</tr>
<tr>
<td>20</td>
<td>2.78</td>
<td>3.49</td>
<td>10</td>
</tr>
<tr>
<td>30</td>
<td>2.76</td>
<td>3.45</td>
<td>9.96</td>
</tr>
<tr>
<td>40</td>
<td>2.76</td>
<td>3.44</td>
<td>9.93</td>
</tr>
<tr>
<td>50</td>
<td>2.75</td>
<td>3.42</td>
<td>10.17</td>
</tr>
<tr>
<td>60</td>
<td>3.36</td>
<td>3.4</td>
<td>12.12</td>
</tr>
<tr>
<td>70</td>
<td>2.73</td>
<td>3.4</td>
<td>10.06</td>
</tr>
<tr>
<td>80</td>
<td>2.72</td>
<td>3.4</td>
<td>10.05</td>
</tr>
</tbody>
</table>

Figure 14 shows the total running time of queries q_2 and q_3 on different data sizes with the fixed bandwidth of 56KB/sec. The total running time includes the runtimes for all the steps in 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, the resulting time savings are significant. For instance, for query q_3 and a 70MB database, DECOMP took 125 seconds, ZIP took 103 seconds, and DECOMP&ZIP took only 50 seconds.

![Figure 14: Runtimes for DECOMP, ZIP, DECOMP&ZIP for varying data sizes.](image_url)
In another series of experiments, we fixed the database size to 60MB and let the bandwidth vary. Figure 15 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 decreases as the network speed goes up.

![Figure 15: Runtimes for DECOMP, ZIP, DECOMP&ZIP for varying bandwidths.](image)

**Summary**: Our experiments show that our proposed approach can reduce the data-communication size significantly when the query result has 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 total running time of sending the query result to the client.

7 Conclusions

In this paper we studied the problem of finding viewsets to answer a workload of conjunctive queries, such that the total size of the views is minimal. We gave decidability and complexity results for workloads of conjunctive queries; the results differ significantly depending on whether the queries have self-joins. We studied the problem for a single query without self-joins in client-server environments, and developed techniques to compute a view decomposition plan efficiently. Our experiments in this case showed that the techniques can significantly reduce the communication costs of transferring the answers to large-join queries from the server to the client.

References


8 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 1_extendedprice, 1_discount, 1_quantity, 1_orderkey, 1_lineitem, n_name as nation, year(o_orderdate) as o_year, ps_supplycost
FROM lineitem, nation, orders, part, partsupp, supplier
WHERE 1_quantity > 30
  AND 1_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, 1_extendedprice, 1_discount, 1_orderkey, l_lineitem, n_name
FROM customer, lineitem, nation, orders
WHERE c_custkey = o_custkey
  AND c_nationkey = n_nationkey
  AND 1_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, 1_quantity, 1_extendedprice, 1_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 = 1_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_totalprice, o_comment,
1_extendedprice, 1_discount
from customer, orders, lineitem,
supplier, nation, region
where c_custkey = o_custkey
and 1_orderkey = o_orderkey
and 1_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
1_extendedprice*(1-1_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 1_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_suppkey
and ps_partkey = 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'