Optimizing Large Join Queries in Mediation Systems

Ramana Yerneni, Chen Li
Jeffrey Ullman, Hector Garcia-Molina

{yerneni, chenli, ullman, hector}@cs.stanford.edu, Stanford University, USA

Abstract. In data integration systems, queries posed to a mediator need to be translated into a sequence of queries to the underlying data sources. In a heterogeneous environment, with sources of diverse and limited query capabilities, not all the translations are feasible. In this paper, we study the problem of finding feasible and efficient query plans for mediator systems. We consider conjunctive queries on mediators and model the source capabilities through attribute-binding adornments. We use a simple cost model that focuses on the major costs in mediation systems, those involved with sending queries to sources and getting answers back. Under this metric, we develop two algorithms for source query sequencing – one based on a simple greedy strategy and another based on a partitioning scheme. The first algorithm produces optimal plans in some scenarios, and we show a linear bound on its worst case performance when it misses optimal plans. The second algorithm generates optimal plans in more scenarios, while having no bound on the margin by which it misses the optimal plans. We also report on the results of the experiments that study the performance of the two algorithms.

1 Introduction

Integration systems based on a *mediation* architecture [31] provide users with seamless access to data from many heterogeneous sources. Examples of such systems are TSIMMIS [3], Garlic [8], Information Manifold [14], and DISCO [26]. In these systems, mediators define integrated views based on the data provided by the sources. They translate user queries on integrated views into source queries and postprocessing operations on the source query results. The translation process can be quite challenging when integrating a large number of heterogeneous sources.

One of the important challenges for integration systems is to deal with the diverse capabilities of sources in answering queries [8, 15, 19]. This problem arises due to the heterogeneity in sources ranging from simple file systems to full-fledged relational databases. The problem we address in this paper is how to generate efficient mediator query plans that respect the limited and diverse capabilities of data sources. In particular, we focus our attention on the kind of queries that are the most expensive in mediation systems, *large join queries*. We propose efficient algorithms to find good plans for such queries.

1.1 Cost Model

In many applications, the cost of query processing in mediator systems is dominated by the cost of interacting with the sources. Hence, we focus on the costs associated with sending queries to sources. Our results are first stated using a very simple cost model where we count the total number of source queries in a plan as its cost. In spite of the simplicity of the cost model, the optimization problem we are dealing with remains NP-hard. Later in the paper, we show how to extend our main results to a more complex cost model that charges a fixed cost per query plus a variable cost that is proportional to the amount of data transferred.

1.2 Capabilities-Based Plan Generation

We consider mediator systems, where users pose conjunctive queries over integrated views provided by the mediator. These queries are translated into conjunctive queries over the source views to arrive at *logical query plans*. The logical plans deal only with the content descriptions of the sources. That is, they tell the mediator which sources provide the relevant data and what postprocessing operations need to be performed on this data. The logical plans are later translated into *physical plans* that specify details such as the order in which the sources are contacted and the exact queries to be sent. Our goal in this paper is to develop algorithms that will translate a mediator logical plan into an efficient, *feasible* (does not exceed the source capabilities) physical plan. We illustrate the process of translating a logical plan into a physical plan by an example.

Example 1. Consider three sources that provide information about movies, and a mediator that provides an integrated view:

Source	Contents	Must Bind
S_1	`R(studio, title)`	either `studio` or `title`
S_2	`S(title, year)`	`title`
S_3	`T(title, stars)`	`title`

Mediator View:

```
Movie(studio,title,year,stars) :-
          R(studio,title), S(title,year), T(title,stars)
```

The "Must Bind" column indicates what attributes must be specified at a source. For instance, queries sent to S_1 must either provide the `title` or the `studio`. Suppose the user asks for the titles of all movies produced by Paramount in 1955 in which Gregory Peck starred. That is,

```
ans(title) :- Movie('Paramount', title, '1955', 'Gregory Peck')
```

The mediator would translate this query to the logical plan:

```
ans(title) :- R('Paramount', title), S(title, '1955'), T(title,
                      'Gregory Peck')
```

The logical plan states the information the mediator needs to obtain from the three sources and how it needs to postprocess this information. In this example, the mediator needs to join the results of the three source queries on the `title` attribute. There are many physical plans that correspond to this logical plan (based on various join orders and join methods). Some of these plans are feasible while others are not. Here are two physical plans for this logical plan:

- Plan P_1: Send query `R('Paramount', title)` to S_1; send query `S(title, '1955')` to S_2; and send query `T(title, 'Gregory Peck')` to S_3. Join the results of the three source queries on the `title` attribute and return the `title` values to the user.
- Plan P_2: Get the titles of movies produced by Paramount from source S_1. For each returned title t, send a query to S_2 to get its `year` and check if it is '1955.' If so, send a query to S_3 to get the `stars` of movie t. If the set of `stars` contains 'Gregory Peck,' return t to the user.

In the above plans, the first one is not feasible because the queries to sources S_2 and S_3 do not provide a binding for `title`. The second one is feasible. There are actually other feasible plans (for instance, we can reverse the S_2 and S_3 queries of P_2). If P_2 is the cheapest feasible plan, the mediator may execute it.

As illustrated by the example, we need to solve the following problem: Given a logical plan and the description of the source capabilities, find feasible physical plans for the logical plan. The central problem is to determine the evaluation order for logical plan subgoals, so that attributes are appropriately bound. Among all the feasible physical plans, pick the most efficient one.

1.3 Related Work

The problem of ordering subgoals to find the best feasible sequence can be viewed as the well known *join-order* problem. More precisely, we can assign *infinite* cost to infeasible sequences and then find the best join order.

The join-order problem has been extensively studied in the literature, and many solutions have been proposed. Some solutions perform a rather exhaustive enumeration of plans, and hence do not scale well [1, 2, 4, 5, 8, 15, 17, 19, 20, 22, 29]. In particular, we are interested in Internet scenarios with many sources and subgoals, so these schemes are too expensive. Some other solutions reduce the search space through techniques like *simulated annealing*, *random probes*, or other heuristics [6, 11, 12, 16, 18, 23–25]. While these approaches may generate efficient plans in some cases, they do not have any performance guarantees in terms of the quality of plans generated (i.e., the plans generated by them can be arbitrarily far from the optimal one). Many of these techniques may even fail to generate a feasible plan, while the user query does have a feasible plan.

The remaining solutions [10, 13, 21] use specific cost models and clever techniques that exploit them to produce optimal join orders efficiently. While these solutions are very good for the join-order problem where those cost models are appropriate, they are hard to adopt in our context because of two difficulties. The first is that it is not clear how to model the feasibility of mediator query plans in their frameworks. A direct application of their algorithms to the problem we are studying may end up generating infeasible plans, when a feasible plan exists. The second difficulty is that when we use cost models that emphasize the main costs in mediator systems, the optimality guarantees of their algorithms may not hold.

1.4 Our Solution

In this paper, we develop two algorithms that find good feasible plans. The first algorithm runs in $O(n^2)$ time, where n is the number of subgoals in the logical plan. We provide a linear bound on the margin by which this algorithm can miss the optimal plan. Our second algorithm can guarantee optimal plans in more scenarios than the first, although there is no bounded optimality for its plans. Both our algorithms are guaranteed to find a feasible plan, if the user query has a feasible plan. Furthermore, we show through experiments that our algorithms have excellent running time profiles in a variety of scenarios, and very often find optimal or close-to-optimal plans. This combination of efficient, scalable algorithms that generate provably good plans is not achieved by previously known approaches.

2 Preliminaries

In this section, we introduce the notation we use throughout the paper. We also discuss the cost model used in our optimization algorithms.

2.1 Source Relations and Logical Plans

Let S_1, ..., S_m be m sources in an integration system. To simplify the presentation, we assume that sources provide their data in the form of relations. If sources have other data models, one could use *wrappers* [9] to create the simple relational view of data. Each source is assumed to provide a single relation. If a source provides multiple relations, we can model it in our framework as a set of logical sources, all having the same physical source. Example 1 showed three sources S_1, S_2 and S_3 providing three relations R, S and T respectively.

A *query* to a source specifies atomic values to a subset of the source relation attributes and obtains the corresponding set of tuples. A source supports a set of access templates on its relation that specify binding adornment requirements for source queries.[1] In Example 1, source S_2 had one access template:

[1] We consider source-capabilities described as *bf* adornment patterns that distinguish bound (*b*) and free (*f*) argument positions [27]. The techniques developed in this paper can also be employed to solve the problem of mediator query planning when other source capability description languages are used.

$S^{bf}(\texttt{title}, \texttt{year})$, while source S_1 had two access templates: $R^{bf}(\texttt{studio}, \texttt{title})$ and $R^{fb}(\texttt{studio}, \texttt{title})$.

User queries to the mediator are conjunctive queries on the integrated views provided by the mediator. Each integrated view is defined as a set of conjunctive queries over the source relations. The user query is translated into a logical plan, which is a set of conjunctive queries on the source relations. The answer to the user query is the union of the results of this set of conjunctive queries. Example 1 showed a user query that was a conjunctive query over the *Movie* view, and it was translated into a conjunctive query over the source relations.

In order to find the best feasible plan for a user query, we assume that the mediator processes the logical plan one conjunctive query at a time (as in [3, 8, 14]). Thus, we reduce the problem of finding the best feasible plan for the user query to the problem of finding the best feasible plan for a conjunctive query in the logical plan. In a way, from now on, we assume without loss of generality that a logical plan has a single conjunctive query over the source relations.

Let the logical plan be $H :- C_1, C_2, \ldots, C_n$. We call each C_i a *subgoal*. Each subgoal specifies a query on one of the source relations by binding a subset of the attributes of the source relation. We refer to the attributes of subgoals in the logical plan as *variables*. In Example 1, the logical plan had three subgoals with four variables, three of which were bound.

2.2 Binding Relations and Source Queries

Given a sequence of n subgoals $C_1, C_2, \ldots, C_n$, we define a corresponding sequence of $n+1$ *binding relations* $I_0, I_1, \ldots, I_n$. I_0 has as its schema the set of variables bound in the logical plan, and it has a single tuple, denoting the bindings specified in the logical plan. The schema of I_1 is the union of the schema of I_0 and the schema of the source relation of C_1. Its instance is the join of I_0 and the source relation of C_1. Similarly, we define I_2 in terms of I_1 and the source relation of C_2, and so on. The answer to the conjunctive query is defined by a projection operation on I_n.

In order to compute a binding relation I_j, we need to join I_{j-1} with C_j. There are two ways to perform this operation:

1. Use I_0 to send a query to the source of C_j (by binding a subset of its attributes); perform the join of the result of this source query with I_{j-1} at the mediator to obtain I_j.
2. For $j \geq 2$, use I_{j-1} to send a set of queries to the source relation of C_j (by binding a subset of its attributes); union the results of these source queries; perform the join of this union relation with I_{j-1} to obtain I_j.

We call the first kind of source query a *block query* and the second kind a *parameterized query*[2]. Obviously, answering C_j through the first method takes a single source query, while answering it by the second method can take many

[2] A parameterized query is different from a *semijoin* where one can send multiple bindings for an attribute in a single query.

source queries. The main reason why we need to consider parameterized queries is that it may not be possible to answer some of the subgoals in the logical plan through block queries. This may be because the access templates for the corresponding source relations require bindings of variables that are not available in the logical plan. In order to answer these subgoals, we must use parameterized queries by executing other subgoals and collecting bindings for the required parameters of C_j.

2.3 The Plan Space

The space of all possible plans for a given user query is defined first by considering all sequences of subgoals in its logical plan. In a sequence, we must then decide on the choice of queries for each subgoal (among the set of block queries and parameterized queries available for the subgoal). We call a plan in this space *feasible* if all the queries in it are answerable by the sources. Note that the number of feasible physical plans, as well as the number of all plans, for a given logical plan can be exponential in the number of subgoals in the logical plan.

Note that the space of plans we consider is similar to the space of left-deep-tree executions of a join query. As stated in the following theorem, we do not miss feasible plans by not considering bushy-tree executions.

Theorem 1. *We do not miss feasible plans because of considering only left-deep-tree executions of the subgoals.*

Proof. For any feasible execution of the logical plan based on a bushy tree of subgoals, we can construct another feasible execution based on a left-deep tree of subgoals (with the same leaf order). This is similar to the *bound-is-easier* assumption of [28]. See the full version of our paper [32] for a detailed proof.

2.4 The Formal Cost Model

Our cost model is defined as follows:

1. The cost of a subgoal in the feasible plan is the number of source queries needed to answer this subgoal.
2. The cost of a feasible plan is the sum of the costs of all the subgoals in the plan.

We develop the main results of the paper in the simple cost model presented above. Later, in Section 5, we will show how to extend these results to more complex cost models. We also consider more practical cost models in Section 6 where we analyze the performance of our algorithms. Here, we note that even in the simple cost model that counts only the number of source queries, the problem of finding the optimal feasible plan is quite hard.

Theorem 2. *The problem of finding the feasible plan with the minimum number of source queries is NP-hard.*

Proof. We reduce the Vertex Cover problem ([7]) to our problem. Since the Vertex Cover problem is NP-complete, our problem is NP-hard.

Given a graph G with n vertices $V_1, \ldots, V_n$, we construct a database and a logical plan as follows. Corresponding to each vertex V_i we define a relation R_i. For all $1 \leq i \leq j \leq n$, if V_i and V_j are connected by an edge in G, R_i and R_j include the attribute A_{ij}. In addition, we define a special attribute X and two special relations R_0 and R_{n+1}. In all, we have a total of $m+1$ attributes, where m is the number of edges in G. The special attribute X is in the schema of all the relations. The special relation R_{n+1} also has all the attributes A_{ij}. That is, R_0 has only one attribute and R_{n+1} has $m+1$ attributes. Each relation has a tuple with a value of 1 for each of its attributes. In addition, all relations except R_{n+1} include a second tuple with a value of 2 for all their attributes. Each relation has a single access template: R_0 has no binding requirements, R_1 through R_n require the attribute X to be bound, and R_{n+1} requires all of the attributes to be bound. Finally, the logical plan consists of all the $n+2$ relations, with no variables bound.

It is obvious that the above construction of the database and the logical plan takes time that is polynomial in the size of G. Now, we show that G has a vertex cover of size k if and only if the logical plan has a feasible physical plan that requires $(n+k+3)$ source queries.

Suppose G has a vertex cover of size k. Without loss of generality, let it be $V_1, \ldots, V_k$. Consider the physical plan P that first answers the subgoal R_0 with a block query, then answers $R_1, \ldots, R_k, R_{n+1}, R_{k+1}, \ldots, R_n$ using parameterized queries. P is a feasible plan because R_0 has no binding requirements, $R_1, \ldots, R_k$ need X to be bound and X is available from R_0, and $R_1, \ldots, R_k$ will bind all the variables (since $V_1, \ldots, V_k$ is a vertex cover). In P, R_0 is answered by a single source query, $R_1, \ldots, R_k$ and R_{n+1} are answered by two source queries each, and $R_{k+1}, \ldots, R_n$ are answered by one source query each. This gives a total of $(n+k+3)$ source queries for this plan. Thus, we see that if G has a vertex cover of size k, we have a feasible plan with $(n+k+3)$ source queries.

Suppose, there is a feasible plan P' with f source queries. In P', the first relation must be R_0, and this subgoal must be answered by a block query (because the logical plan does not bind any variables). All the other subgoals must be answered by parameterized queries. Consider the set of subgoals in P' that are answered before R_{n+1} is answered. Let j be the size of this set of subgoals (excluding R_0). Since R_{n+1} needs all attributes to be bound, the union of the schemas of these j subgoals must be the entire attribute set. That is, the vertices corresponding to these j subgoals form a vertex cover in G. In P', each of these j subgoals takes two source queries, along with R_{n+1}, while the rest of $(n-j)$ subgoals in $R_1, \ldots, R_n$ take one source query each. That is, $f = 1+2*j+2+(n-j)$. From this, we see that we can find a vertex cover for G of size $(f-n-3)$.

Hence, G has a vertex cover of size k if and only if there is a feasible plan with $(n+k+3)$ source queries. That is, we have reduced the problem of finding the minimum vertex cover in a graph to our problem of finding a feasible plan with minimum source queries.

In our cost model, it turns out that it is safe to restrict the space of plans to those based on left-deep-tree executions of the set of subgoals.

Theorem 3. *We do not miss the optimal plan by not considering the executions of the logical plan based on bushy trees of subgoals.*

Proof. See the full version of our paper [32] for the proof.

3 The CHAIN Algorithm

In this section, we present the CHAIN algorithm for finding the best feasible query plan. This algorithm is based on a greedy strategy of building a single sequence of subgoals that is feasible and efficient.

The CHAIN Algorithm
Input: Logical plan – subgoals and bound variables.
Output: Feasible physical plan.

```
• Initialize:
        S ← {C_1, C_2, ..., C_n}  /*set of subgoals in the logical plan*/
        B ← set of bound variables in the logical plan
        L ← φ    /* start with an empty sequence */
• Construct the sequence of subgoals:
      while (S ≠ φ ) do
        M ← infinity;
        N ← null;
        for each subgoal C_i in S do    /* find the cheapest subgoal */
          if (C_i is answerable with B) then
            c ← Cost_L(C_i);        /* get the cost of this subgoal in sequence L */
            if ( c < M ) then
              M ← c;
              N ← C_i;
        /* If no next answerable subgoal, declare no feasible plan */
        if (N = null)
          return(φ);
        /* Add next subgoal to plan */
        L ← L + N;
        S ← S − {N};
        B ← B ∪ {variables of N};
• Return the feasible plan:
      return(Plan(L));                  /* construct plan from sequence L */
```

Fig. 1. Algorithm CHAIN

As shown in Figure 1, CHAIN starts by finding all subgoals that are answerable with the initial bindings in the logical plan. It then picks the answerable subgoal with the least cost and computes the additional variables that are now bound due to the chosen subgoal. It repeats the process of finding answerable subgoals, picking the cheapest among them and updating the set of bound variables, until no more subgoals are left or some subgoals are left but none of them is answerable. If there are subgoals left over, CHAIN declares that there is no feasible plan; otherwise it outputs the plan it has constructed.

3.1 Complexity and Optimality of CHAIN

Here we demonstrate: the CHAIN algorithm is very efficient; it is guaranteed to find feasible plans when they exist; and there is a linear bound on the optimality of the plans it generates. Due to space limitations, we have not provided proofs for all the lemmas and theorems. They are available in the full version of the paper [32].

Lemma 1. *CHAIN runs in $O(n^2)$ time, where n is the number of subgoals.* [3]

Lemma 2. *CHAIN will generate a feasible plan, if the logical plan has feasible physical plans.*

Lemma 3. *If the result of the user query is nonempty, and the number of subgoals in the logical plan is less than 3, CHAIN is guaranteed to find the optimal plan.*

Lemma 4. *CHAIN can miss the optimal plan if the logical plan has more than 2 subgoals.*

Proof. We construct a logical plan with 3 subgoals and a database instance that result in CHAIN generating a suboptimal plan.

$R^{bff}(A,B,D)$	$S^{bf}(B,E)$	$T^{bf}(D,F)$
(1, 1, 1)	(1, 1)	(4, 1)
(1, 2, 2)	(2, 1)	(5, 1)
(1, 3, 3)	(3, 1)	(6, 1)
(1, 1, 4)	(4, 1)	(7, 1)

Table 1. Database Instance for Lemma 4

Consider a logical plan $H :- R(1,B,D),\ S(B,E),\ T(D,F)$ and the database instance shown in Table 1. For this logical plan and database, CHAIN will generate the plan: $R \rightarrow S \rightarrow T$, with a total cost of $1+3+4=8$. We observe that a cheaper feasible plan is: $R \rightarrow T \rightarrow S$, with a total cost of $1+4+1=6$. Thus, CHAIN misses the optimal plan in this case.

[3] We are assuming here that finding the cost of a subgoal following a partial sequence takes $O(1)$ time.

It is not difficult to find situations in which the CHAIN algorithm misses the optimal plan. However, surprisingly, there is a linear upper bound on how far its plan can be from the optimal. In fact, we prove a stronger result in Lemma 5.

Lemma 5. *Suppose P^c is the plan generated by CHAIN for a logical plan with n subgoals; P^o is the optimal plan, and E_{max} is the cost of the most expensive subgoal in P^o. Then,*

$$Cost(P^c) \leq n \times E_{max}$$

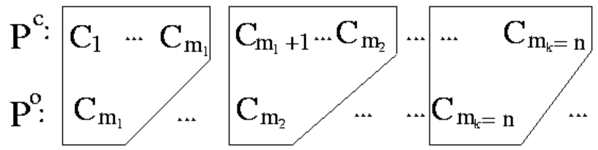

Fig. 2. Proof for Lemma 5

Proof. Without loss of generality, suppose the sequence of subgoals in P^c is $C_1, C_2, \ldots, C_n$. As shown in Figure 2, let the first subgoal in P^o be C_{m_1}. Let G_1 be the prefix of P^c, such that $G_1 = C_1 \ldots C_{m_1}$. When CHAIN chooses C_1, the subgoal C_{m_1} is also answerable. This implies that the cost of C_1 in P^c is less than or equal to the cost of C_{m_1} in P^o. After processing C_1 in P^c, the subgoal C_{m_1} remains answerable and its cost of processing cannot increase. So, if CHAIN has chosen another subgoal C_2 instead of C_{m_1}, once again we can conclude that the cost of C_2 in P^c is not greater than the cost of C_{m_1} in P^o. Finally, at the end of G_1, when C_{m_1} is processed in P^c, we note that the cost of C_{m_1} in P^c is no more than the cost of C_{m_1} in P^o. Thus, the cost of each subgoal of G_1 is less than or equal to the cost of C_{m_1} in P^o.

We call C_{m_1} the first *pivot* in P^o. We define the next pivot C_{m_2} in P^o as follows. C_{m_2} is the first subgoal after C_{m_1} in P^o such that C_{m_2} is not in G_1. Now, we can define the next subsequence G_2 of P^c such that the last subgoal of G_2 is C_{m_2}. The cost of each subgoal in G_2 is less than or equal to the cost of C_{m_2}.

We continue finding the rest of the pivots $C_{m_3}, \ldots, C_{m_k}$ in P^o and the corresponding subsequences $G_3, \ldots, G_k$ in P^c. Based on the above argument, we have

$$\forall C_i \in G_j : (\text{cost of } C_i \text{ in } P^c) \leq (\text{cost of } C_{m_j} \text{ in } P^o)$$

From this, it follows that

$$Cost(P^c) = \sum_{j=1}^{k} \sum_{C_i \in G_j} (\text{cost of } C_i \text{ in } P^c) \leq \sum_{j=1}^{k} |G_j| \times (\text{cost of } C_{m_j} \text{ in } P^o) \leq n \times E_{max}$$

Theorem 4. *CHAIN is n-competitive. That is, the plan generated by CHAIN can be at most n times as expensive as the optimal plan, where n is the number of subgoals.*

Proof. Follows from Lemma 5.

The cost of the plan generated by CHAIN can be arbitrarily close to the cost of the optimal plan multiplied by the number of subgoals; i.e., Theorem 4 cannot be improved. However, in many situations CHAIN yields optimal plans or plans whose cost is very close to that of the optimal plan as demonstrated in Section 6.

4 The PARTITION Algorithm

In this section, we present another algorithm called *PARTITION* for finding efficient feasible plans. PARTITION takes a very different approach to solve the plan generation problem. It is guaranteed to generate optimal plans in more scenarios than CHAIN but has a worse running time.

4.1 PARTITION

The formal description of PARTITION is available in the full version of the paper [32]. Here, we present the essential aspects of the algorithm.

The PARTITION algorithm has two phases. In the first phase, it organizes the subgoals into *clusters* based on the capabilities of the sources. The property satisfied by the clusters generated by the first phase of PARTITION is as follows. All the subgoals in the first cluster are answerable by block queries; all the subgoals in each subsequent cluster are answerable by parameterized queries that use attribute bindings from the subgoals of the earlier clusters. To obtain the clusters, PARTITION keeps track of the set of bound variables V. Initially, V is the set of variables bound in the logical plan. The first phase of PARTITION is divided into many rounds (one per cluster). In each round, the set of answerable subgoals based on the bound variable set V is collected into a new cluster. These subgoals are removed from the set of subgoals that are yet to be picked, and the variables bound by these subgoals are added to V. If in a round of the first phase there are subgoals yet to be picked and none of them is answerable, PARTITION declares that there is no feasible plan for the user query.

In the second phase, PARTITION finds the best subplan for each cluster of subgoals and combines the subplans to arrive at the best overall plan for the user query. The subplan for each cluster is found by enumerating all the sequences of subgoals in the cluster and choosing the one with the least cost.

4.2 Optimality and Complexity of PARTITION

Like the CHAIN algorithm, the PARTITION algorithm always finds feasible plans when they exist. It is guaranteed to find optimal plans in more scenarios than CHAIN. However, when it misses the optimal plans, it can miss them by an unbounded margin. It is also much less efficient than CHAIN, and can take time that is exponential in the number of subgoals of the logical plan. These observations are formally stated by the following lemmas (proofs omitted occasionally due to space limitations).

Lemma 6. *If feasible physical plans exist for a given logical plan, PARTITION is guaranteed to find a feasible plan.*

Lemma 7. *If there are fewer than 3 clusters generated, and the result of the query is nonempty, then PARTITION is guaranteed to find the optimal plan.*

Proof. We proceed by a simple case analysis. There are two cases to consider.

The first case is when there is only one cluster Γ_1. PARTITION finds the best sequence among all the permutations of the subgoals in Γ_1. Since Γ_1 contains all the subgoals of the logical plan, PARTITION will find the best possible sequence.

The second case is when there are two clusters Γ_1 and Γ_2. Let P be the optimal feasible plan. We will show how we can transform P into a plan in the plan space of PARTITION that is at least as good as P.

Let C_i be a subgoal in Γ_1. There are two possibilities: (a) C_i is answered in P by using a block query; (b) C_i is answered in P by using parameterized queries. If C_i is answered by a block query, we make no change to P. Otherwise, we modify P as follows. As the result of the query is not empty, the cost of subgoal C_i (using parameterized queries) in P must be at least 1. Since C_i is in the first cluster, it can be answered by using a block query. So we can modify P by replacing the parameterized queries for C_i with the block query for C_i. Since the cost of a block query can be at most 1, this modification cannot increase the cost of P. For all subgoals in Γ_1, we repeat the above transformation until we get a plan P', in which all the subgoals in Γ_1 are answered by using block queries.

We apply a second transformation to P' with respect to the subgoals in Γ_1. Since all these subgoals are answered by block queries in P', we can move them to the beginning of P' to arrive at a new plan P''. Moving these subgoals ahead of the other subgoals will preserve the feasibility of the plan. It is also true that this transformation cannot increase the cost of the plan. This is because it does not change the cost of these subgoals, and it cannot increase the cost of the other subgoals in the sequence. Hence, P'' cannot be more expensive than P'.

After the two-step transformation, we get a plan P'' that is as good as P. Finally, we note that P'' is in the plan space of PARTITION, and so the plan generated by PARTITION cannot be worse than P''. Thus, the plan found by PARTITION must be as good as the optimal plan.

Lemma 8. *If the number of subgoals in the logical plan does not exceed 3, and the result of the query is not empty, then PARTITION will always find the optimal plan.*

Proof. Follows from Lemma 7.

PARTITION cannot generate the optimal plan in many cases. One can construct logical plans with as few as 4 subgoals that lead the algorithm to generate a sub-optimal plan. Also, PARTITION can miss the optimal plan by a margin that is unbounded by the query parameters.

Lemma 9. *For any integer $m > 0$, there exists a logical plan and a database for which PARTITION generates a plan that is at least m times as expensive as the optimal plan.*

Proof. Refer to [32] for the detailed proof. The essential idea is to construct a logical plan and a database for any given m that will make PARTITION miss the optimal plan by a factor greater than m.

Lemma 10. *The PARTITION algorithm runs in $O(n^2+(k_1!+k_2!+\ldots+k_p!))$, where n is the number of subgoals in the logical plan, p is the number of clusters found by PARTITION and k_i is the number of subgoals in the i^{th} cluster.* [4]

4.3 Variations of PARTITION

We have seen that the PARTITION algorithm can miss the optimal plan in many scenarios, and in the worst case it has a running time that is exponential in the number of subgoals in the logical plan. In a way, it attempts to strike a balance between running time and the ability to find optimal plans. A naive algorithm that enumerates all sequences of subgoals will always find the optimal plan, but it may take much longer than PARTITION. PARTITION tries to cut down on the running time, and gives up the ability to find optimal plans to a certain extent. Here, we consider two variations of PARTITION that highlight this trade-off.

We call the first variation FILTER. This variation is based on the observation of Lemma 7. FILTER also has two phases like PARTITION. In its first phase, it mimics PARTITION to arrive at the clusters $\Gamma_1, \Gamma_2, \ldots, \Gamma_p$. At the end of the first phase, it keeps the first cluster as is, and collapses all the other clusters into a new second cluster Γ'. That is, it ends up with Γ_1 and Γ'. The second phase of FILTER is identical to that of PARTITION. FILTER is guaranteed to find the optimal plan (as long as the query result is nonempty), but its running time is much worse than PARTITION. Yet, it is more efficient than the naive algorithm that enumerates all plans.

Lemma 11. *If the user query has nonempty result, FILTER will generate the optimal plan.*

Proof. We can prove this lemma in the same way we proved Lemma 7.

Lemma 12. *The running time of FILTER is $O(n^2 + (k_1! + (n - k_1)!)$.* [5]

The second variation of PARTITION is called SCAN. This variation focuses on efficient plan generation. The main idea here is to simplify the second phase

[4] If the query result in nonempty, PARTITION can consider just one sequence (instead of $k_1!$) for the first cluster.

[5] If the query result in nonempty, FILTER can consider just one sequence (instead of $k_1!$) for the first cluster.

of PARTITION so that it can run efficiently. The penalty is that SCAN may not generate optimal plans in many cases where PARTITION does.

SCAN also has two phases of processing. The first phase is identical to that of PARTITION. In the second phase, SCAN picks an arbitrary order for each cluster without searching over all the possible orders. This leads to a second phase that runs in $O(n)$ time. Note that since it does not search over the space of subsequences for each cluster, SCAN tends to generate plans that are inferior to those of PARTITION.

Lemma 13. *SCAN runs in $O(n^2)$ time, where n is the number of subgoals in the logical plan.*

5 Other Cost Models

So far, we discussed algorithms that minimize the number of source queries. Now, we consider more complex cost models where different source queries can have different costs.

First, we consider a simple extension (say M_1) where the cost of a query to source S_i is e_i. That is, queries to different sources cost different amounts. Note that in M_1, we still do not charge for the amount of data transferred. Nevertheless, it is strictly more general than the model we discussed in Section 2. All of our results presented so far hold in this new model.

Theorem 5. *In the cost model M_1, Theorem 4 holds. That is, the CHAIN algorithm is n-competitive, where n is the number of subgoals.*

Theorem 6. *In the cost model M_1, Lemma 7 holds. That is, the PARTITION algorithm will find the optimal plan, if there are at most two clusters and the user query has nonempty result.*

Next, we consider a more complex cost model (say M_2) where the data transfer costs are factored in. That is, the cost of a query to source S_i is $e_i + f_i \times$ (size of query result). Note that this cost model is strictly more general than M_1.

Theorem 7. *In the cost model M_2, Theorem 4 holds. That is, the CHAIN algorithm is n-competitive, where n is the number of subgoals.*

Theorem 8. *In the cost model M_2, Lemma 7 does not hold. That is, the PARTITION algorithm cannot guarantee the optimal plan, even when there are at most two clusters.*

We observe that the n-competitiveness of CHAIN holds in any cost model with the following property: the cost of a subgoal in a plan does not increase by postponing its processing to a later time in the plan. We also note that the PARTITION algorithm with two clusters will always find the optimal plan (assuming the query has nonempty result) if block queries cannot cost more than the corresponding parameterized queries. This property holds, for instance, in model M_1 and not in model M_2. When one considers cost models other than those discussed here, these properties may hold in them and consequently CHAIN and PARTITION may yield very good results.

6 Performance Analysis

In this section, we address the questions: How often do PARTITION and CHAIN find the optimal plan? When they miss the optimal plan, what is the expected margin by which they miss? We answer these questions by experiments in a simulated environment. We used both the simple cost model of Section 2.4 as well as the more complex cost model M_2 of Section 5 in our performance analysis. The results did not deviate much from one cost model to the other. The details of the experiments are in [32]. Here, we briefly mention the important results based on the simpler cost model of Section 2.4.

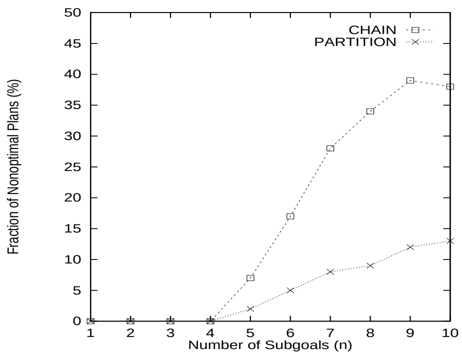

Fig. 3. Rate of Missing Optimal Plans

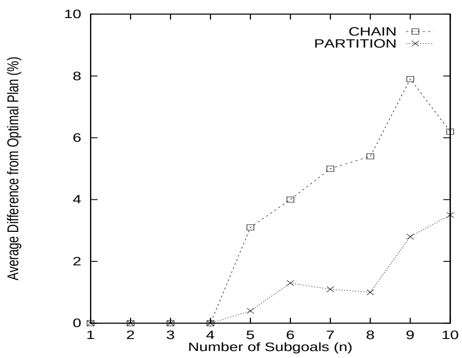

Fig. 4. Margin of Missing Optimal Plans

Figure 3 plots the fraction of the times the algorithms missed the optimal plans vs. number of query subgoals. Over a set of 1000 queries with number of subgoals ranging from 1 to 10, PARTITION generated the optimal plan in more than 95% of the cases, and CHAIN generated the optimal plan more than 75% of the time. This result is surprising because we know that PARTITION can miss optimal plans for queries with as few as 4 subgoals and CHAIN can miss optimal plans for queries with as few as 3 subgoals.

Figure 4 plots the average margin by which generated plans missed the optimal plan vs. the number of query subgoals. Both CHAIN and PARTITION found near-optimal plans over the entire range of queries and, on the average, missed the optimal plan by less than 10%.

In summary, the PARTITION algorithm can have excellent practical performance, even though it gives very few theoretical guarantees. CHAIN also has very good performance, well beyond the theoretical guarantees we proved in Section 3. Finally, comparing the two algorithms, we observe that PARTITION consistently outperforms CHAIN in finding near-optimal plans.

7 Conclusion

In this paper, we considered the problem of query planning in heterogeneous data integration systems based on the mediation approach. We employed a cost model that focuses on the main costs in mediation systems. In this cost model, we developed two algorithms that guarantee the generation of feasible plans (when they exist). We showed that the problem at hand is NP-hard. One of our algorithms runs in polynomial time. It generates optimal plans in many cases and in other cases it has a linear bound on the worst case margin by which it misses the optimal plans. The second algorithm finds optimal plans in more scenarios, but has no bound on the margin of missing the optimal plans in the bad scenarios. We analyzed the performance of our algorithms using simulation experiments and extended our results to more complex cost models.

References

1. P. Apers, A. Hevner, S. Yao. Optimization Algorithms for Distributed Queries. In *IEEE Trans. Software Engineering*, 9(1), 1983.
2. P. Bernstein, N. Goodman, E. Wong, C. Reeve, J. Rothnie. Query Processing in a System for Distributed Databases (SDD-1). In *ACM Trans. Database Systems*, 6(4), 1981.
3. S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *IPSJ*, Japan, 1994.
4. S. Cluet, G. Moerkotte. On the Complexity of Generating Optimal Left-deep Processing Trees with Cross Products. In *ICDT Conference*, 1995.
5. R. Epstein, M. Stonebraker. Analysis of Distributed Database Strategies. In *VLDB Conference*, 1980.
6. C. Galindo-Legaria, A. Pellenkoft, M. Kersten. Fast, Randomized Join Order Selection – Why Use Transformations? In *VLDB Conference*, 1994.
7. M. Garey, D. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, San Francisco, 1979.

8. L. Haas, D. Kossman, E.L. Wimmers, J. Yang. Optimizing queries across diverse data sources. In *VLDB Conference*, 1997.
9. J. Hammer, H. Garcia-Molina, S. Nestorov, R. Yerneni, M. Breunig, V. Vassalos. Template-Based Wrappers in the TSIMMIS System. In *SIGMOD Conference*, 1997.
10. T. Ibaraki, T. Kameda. On the Optimal Nesting Order for Computing N-relational Joins. In *ACM Trans. Database Systems*, 9(3), 1984.
11. Y. Ioannidis, Y. Kang. Randomized Algorithms for Optimizing Large Join Queries. In *SIGMOD Conference*, 1990.
12. Y. Ioannidis, E. Wong. Query Optimization by Simulated Annealing. In *SIGMOD Conference*, 1987.
13. R. Krishnamurthy, H. Boral, C. Zaniolo. Optimization of Non-recursive Queries. In *VLDB Conference*, 1986.
14. A. Levy, A. Rajaraman, J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *VLDB Conference*, 1996.
15. C. Li, R. Yerneni, V. Vassalos, H. Garcia-Molina, Y. Papakonstantinou, J. Ullman, M. Valiveti. Capability Based Mediation in TSIMMIS. In *SIGMOD Conference*, 1998.
16. K. Morris. An algorithm for ordering subgoals in NAIL!. In *ACM PODS*, 1988.
17. K. Ono, G. Lohman. Measuring the Complexity of Join Enumeration in Query Optimization. In *VLDB Conference*, 1990.
18. C. Papadimitriou, K. Steiglitz. Combinatorial Optimization: Algorithms and Complexity. Prentice-Hall, 1982.
19. Y. Papakonstantinou, A. Gupta, L. Haas. Capabilities-based Query Rewriting in Mediator Systems. In *PDIS Conference*, 1996.
20. A. Pellenkoft, C. Galindo-Legaria, M. Kersten. The Complexity of Transformation-Based Join Enumeration. In *VLDB Conference*, 1997.
21. W. Scheufele, G. Moerkotte. On the Comlexity of Generating Optimal Plans with Cartesian Products. In *PODS Conference*, 1997.
22. P. Selinger, M. Adiba. Access Path Selection in Distributed Databases Management Systems. In *Readings in Database Systems.* Edited by M. Stonebraker. Morgan-Kaufman Publishers, 1994.
23. M. Steinbrunn, G. Moerkotte, A. Kemper. Heuristic and Randomized Optimization for the Join Ordering Problem. In *VLDB Journal*, 6(3), 1997.
24. A. Swami. Optimization of Large Join Queries: Combining Heuristic and Combinatorial Techniques. In *SIGMOD Conference*, 1989.
25. A. Swami, A. Gupta. Optimization of Large Join Queries. In *SIGMOD Conference*, 1988.
26. A. Tomasic, L. Raschid, P. Valduriez. Scaling Heterogeneous Databases and the Design of Disco. In *Int. Conf. on Distributed Computing Systems*, 1996.
27. J. Ullman. Principles of Database and Knowledge-base Systems, Volumes I, II. Computer Science Press, Rockville MD.
28. J. Ullman, M. Vardi. The Complexity of Ordering Subgoals. In *ACM PODS*, 1988.
29. B. Vance, D. Maier. Rapid Bushy Join-Order Optimization with Cross Products. In *SIGMOD Conference*, 1996.
30. V. Vassalos, Y. Papakonstantinou. Describing and using query capabilities of heterogeneous sources. In *VLDB Conference*, 1997.
31. G. Wiederhold. Mediators in the Architecture of Future Information Systems. In *IEEE Computer*, 25:38-49, 1992.
32. R. Yerneni, C. Li, J. Ullman, H. Garcia-Molina. Optimizing Large Join Queries in Mediation Systems. http://www-db.stanford.edu/pub/papers/ljq.ps