Joint Optimization of Cost and Coverage of Information Gathering Plans

Zaiqing Nie & Subbarao Kambhampati

Department of Computer Science and Engineering
Arizona State University
Tempe, AZ 85287-5406
Email: {nie,rao}@asu.edu
ASU CSE TR 01-002

Abstract

Existing approaches for optimizing queries in information integration use decoupled strategies–attempting to optimize coverage and cost in two separate phases. Since sources tend to have a variety of access limitations, this type of phased optimization of cost and coverage can unfortunately lead to expensive planning as well as highly inefficient plans.

In this paper we present techniques for joint optimization of cost and coverage of the query plans. Our algorithms search in the space of parallel query plans that support multiple sources for each subgoal conjunct. The refinement of the partial plans takes into account the potential parallelism between source calls, and the binding compatibilities between the sources included in the plan. We start by introducing and motivating our query plan representation, and arguing that our way of searching in the space of parallel plans can improve both the plan generation and plan execution costs compared to existing approaches. We then briefly review how to compute the cost and coverage of a parallel plan. Next, we provide both a System-R style query optimization algorithm as well as a greedy local search algorithm for searching in the space of such query plans. Finally we present an empirical evaluation that demonstrates the flexibility and efficiency afforded by our algorithms in handling cost-coverage tradeoffs, in comparison to the existing approaches.

1 Introduction

With the vast number of autonomous information sources available on the Internet today, users have access to a large variety of data sources. Data integration systems [CGHI94, LRO96, KW96, AHK96, HKWY97, FW97, DG97b, YPGM98, KMA+98, LKG99, IFF+99, PL00] are being developed to provide a uniform interface to a multitude of information sources, query the relevant sources automatically and restructure the information from different sources.

Query optimization in the context of integrating heterogeneous data sources on the Internet has thus received significant attention of late. It differs from the traditional query optimization in several important ways. To begin with, a traditional optimizer is at the site of the (single) database, and can naturally assume that each relation mentioned in a query is is stored in the same primary database, and that the relation can always be accessed "in-whole." In contrast, in information integration scenarios: (i) the optimizer sits at the mediator (ii) relations are effectively stored across multiple and potentially overlapping sources, each of which may only contain a partial extension of the relation. (iii) sources have a variety of access limitations–in terms of binding pattern restrictions on feasible calls and in terms of the number of disjunctive tuples that can be passed in a single query. (iv) users may have differing objectives in terms of what coverage they want and how much execution cost they are willing to bear for achieving the desired coverage.

Consequently, selecting optimal plans in information integration requires the ability to consider the coverages offered by various sources, and form a query plan with the combination of sources that is estimated to be the best plan given the cost-coverage tradeoffs of the user. The cost of a plan depends on the specific access calls made to the information sources, which in turn depends on the way the various access restrictions on the sources are satisfied. The coverage of the plan depends on the coverages of the individual sources that are accessed as part of the plan. While coverage can be increased by accessing all accessible sources, this leads to plans that are too costly.

Existing approaches for optimizing queries in information integration [LRO96; NLF99; DL99; BRV98; MRV00; PL00] use decoupled strategies–attempting to optimize coverage and cost in two separate phases. Specifically, they first generate a set of feasible "linear plans," that contain at most one source for each query conjunct, and then rank these linear plans in terms of the expected coverage offered by them. Finally top N plans are selected and executed. Since sources tend to have a variety of access limitations, this type of phased optimization of cost and coverage can unfortunately lead to significantly costly planning *and* inef-

ficient plans.

In this paper we present techniques for joint optimization of cost and coverage of the query plans. Our algorithms search in the space of "parallel" query plans that support parallel access to multiple sources for each subgoal conjunct. The generation of the partial plans takes into account the potential parallelism between source calls, and the binding compatibilities between the sources included in the plan. We start, in Section 2, by describing an example scenario that motivates the need for joint optimization of cost and coverage of query plans in information integration. We then argue that our way of searching in the space of parallel query plans, using cost models that combine execution cost and the coverage of the candidate plans, provides a promising approach that reduces both plan generation costs and the plan execution costs. Section 3 discusses the syntax and semantics of the parallel information gathering plans. Section 4 reviews the statistical models we use in our implementation to estimate cost and coverage of a parallel plan, and also discusses the specific methodology we use to combine them into a single utility metric. Section 5 discusses conditions for the existence of single parallel plans of maximal utility. Section 6 describes two algorithms to generate parallel query plans. The first is a System-R style dynamic programming algorithm, while the second is a greedy algorithm.

Section 7 presents an empirical evaluation that shows given a coverage requirement, the plans generated by our approach are significantly better, both in terms of planning cost, and in terms of execution cost, compared to the existing approaches that use phased optimization using linear plans. Specifically, we shall show that the query plans returned by our algorithm give over 80% of the coverage given by the exhaustive enumeration approach in [LRO96], while incurring only 2% of the execution cost incurred by the latter. Moreover, our algorithms incur much less planning time than the enumeration approach in [LRO96], and are able to handle a spectrum of cost-coverage tradeoffs.

2 Motivating Example

Consider a simple mediator that integrates several sources that export information about books. Suppose there are three relations in the global schema of this system: **book**(*ISBN, title, author*), **price-of**(*ISBN, retail-price*), **review-of**(*ISBN, reviewer, review*). Suppose the system can access three sources $\mathbf{S}_{11}$, $\mathbf{S}_{12}$, $\mathbf{S}_{13}$ each of which contain tuples for the **book** relation, two sources $\mathbf{S}_{21}$, $\mathbf{S}_{22}$ each of which contain tuples for the **price-of** relation, and two sources $\mathbf{S}_{31}$, $\mathbf{S}_{32}$ each of which contain tuples for the **review-of** relation. Individual sources differ in the amount of coverage they offer on the relation they export. Table 1 lists some representative statistics for these sources (more detailed description of statistics will be given in Section 4). We will assume that the coverage is measured in terms of the fraction of tuples of the relation in the global schema which are stored in the source relation, and cost is specified in terms of the average response time for a single source call. The last column lists the attributes that must be bound in each call to the source. To simplify matters, let us assume that the sources are "independent" in their coverage (in that the probability that a tuple is present in a given source is independent of the probability that the same tuple is present in another source). Consider the example query:

Q(*title,retail-price,review*) : −

book(*ISBN, title, author*),
price-of(*ISBN, retail-price*),
review-of(*ISBN, reviewer, review*),
title="Data Warehousing",
retail-price<$40.

Bucket Algorithm [LRO96]: The bucket algorithm by Levy et al. [LRO96] will generate three buckets for the three subgoals in the query:

Bucket B(for **book**): $\mathbf{S}_{11}$, $\mathbf{S}_{12}$, $\mathbf{S}_{13}$
Bucket P(for **price-of**): $\mathbf{S}_{21}$, $\mathbf{S}_{22}$
Bucket R(for **review-of**): $\mathbf{S}_{31}$, $\mathbf{S}_{32}$

Once the buckets are generated, the algorithm will enumerate 12 possible plans($= 3 \times 2 \times 2$) corresponding to the selection of one source from each bucket. For each combination, the correctness of the plan is checked, and executable orderings for each plan are computed. Note that the 6 plans that include the source S_{32} are not going to lead to any executable orderings since there is no way of binding the "reviewer" attribute as the input to the source query. Consequently, the set of plans output by the bucket algorithms are:

$$\begin{aligned}
p_1 &= (S_{11}^{fbf} \bowtie S_{21}^{bf}) \bowtie S_{31}^{bff},\\
p_2 &= (S_{11}^{fbf} \bowtie S_{22}^{bf}) \bowtie S_{31}^{bff},\\
p_3 &= (S_{12}^{fbf} \bowtie S_{21}^{bf}) \bowtie S_{31}^{bff},\\
p_4 &= (S_{12}^{fbf} \bowtie S_{22}^{bf}) \bowtie S_{31}^{bff},\\
p_5 &= (S_{21}^{fb} \bowtie S_{13}^{bff}) \bowtie S_{31}^{bff},\\
p_6 &= (S_{22}^{fb} \bowtie S_{13}^{bff}) \bowtie S_{31}^{bff}
\end{aligned}$$

where, the superscripts "f" and "b" are used to specify which attributes are bound in each source call. We call these plans "linear plans" in the sense that they contain at most one source for each of the relations mentioned in the query. Once the feasible logical plans are enumerated, the approach in [LRO96] consists of finding "optimal" execution orders for each of the logical plans, and executing all the plans. While this approach is guaranteed to give maximal coverage, it is often prohibitively expensive in terms of both planning and execution cost. In particular, for a query with n subgoals, and a scenario where there are at most m sources in the bucket of any subgoal, the worst case complexity of this approach (in terms of planning time) is $O(m^n n^2)$, as there can be m^n distinct linear plans, and the cost of finding a feasible order for them using the approach in [LRO96] is $O(n^2)$.

Executing top N Plans: More recent work tried to make-up for the prohibitive execution cost of the enumeration strategy used in [LRO96] by first ranking the enumerated plans in the order of their coverage (or more broadly "quality"),

Sources	Contents	Coverage	Cost	Must bind attributes
S_{11}	book(ISBN,title,author)	70%	300	ISBN or title
S_{12}	book(ISBN,title,author)	50%	200	ISBN or title
S_{13}	book(ISBN,title,author)	60%	600	ISBN
S_{21}	price-of(ISBN,retail-price)	75%	300	ISBN or retail-price
S_{22}	price-of(ISBN,retail-price)	70%	260	ISBN or retail-price
S_{31}	review-of(ISBN,reviewer,review)	70%	300	ISBN
S_{32}	review-of(ISBN,reviewer,review)	50%	400	reviewer

Table 1: *Statistics for the sources in the example system*

and then executing top N plans [FKL97], [NLF99], [DL99], for some arbitrarily chosen N. The idea is to identify the specific plans that are likely to have high coverage and execute those first.

In our example, these ranking procedures might rank $p_1 = (S_{11}^{fbf} \bowtie S_{21}^{bf}) \bowtie S_{31}^{bff}$ as the best plan (since all of the sources have the highest coverage among sources in their buckets), and then rank $p_6 = (S_{22}^{fb} \bowtie S_{13}^{bff}) \bowtie S_{31}^{bff}$, as the second best plan as it contains the sources with highest coverages after executing the best plan p_1.

The problem with this type of approach is that *the plans that are ranked highest in terms of coverage may not necessarily provide the best tradeoffs in terms of execution cost*. In our example, suppose source S_{22} stores 1000 tuples with attribute value retail-price less than \$40, then in plan p_6 we have to query S_{13}, the costliest among the accessible sources, thousand times. The total cost of this plan will thus be more than $6*10^5$. In contrast, a lower ranked plan such as p_4 ($= (S_{12}^{fbf} \bowtie S_{22}^{bf}) \bowtie S_{31}^{bff}$) may cost significantly less, while offering coverage that is competitive with that offered by p_6. For example, assuming that source S_{12} maintains 10 independent ISBN values for title="Data Warehousing", the cost of p_4 may be less than 5800. In such a scenario, most users may prefer executing the plan p_4 first instead of p_6 to avoid incurring the high cost of executing plan p_6.

The lesson from the example above is that *if we want to get a plan that can produce higher quality results with limited cost, it is critical to consider execution costs while doing source selection (rather than after the fact)*. In order to take the cost information into account, we have to consider the source-call ordering during planning, since different source-call orders will result in different execution cost for the same logical plan. In other words, we have to consider source-call ordering and source selection at the same time to get a reasonable execution plan in information integration scenarios.

Need for Parallel Plans: Once we recognize that the cost and coverage need to be taken into account together, we argue that it is better to organize the query planning in terms of "which sources should be called for supporting each subgoal" rather than in terms of "which linear plans have the highest cost/quality tradeoffs." To this end, we introduce the notion of a "parallel" plan, which is essentially a sequence of source sets. Each source set corresponds to a subgoal of the query, such that the sources in that set export that subgoal (relation). The sources in individual source sets can be called in parallel (while the different source sets are processed sequentially).

In our continuing example, looking at the 6 plans generated by the bucket algorithm we can see that the first four plans p_1, p_2, p_3 and p_4 have the same subgoal order (the order of the subgoals of the sources in the plan): book →price-of → review-of, while the other two plans p_5, p_6 have the same subgoal order: price-of →book → review-of. So we can use the following two parallel plans to give all of the tuples that six plans give in this example:

$$p_1' = \big((S_{11}^{fbf} \cup S_{12}^{fbf}) \bowtie (S_{21}^{bf} \cup S_{22}^{bf})\big) \bowtie S_{31}^{bff},$$
$$p_2' = \big((S_{21}^{fb} \cup S_{22}^{fb}) \bowtie S_{13}^{bff}\big) \bowtie S_{31}^{bff}$$

These plans access all the sources related to a given subgoal of the query in parallel (see Section 3.2 for a more detailed description of their semantics). An important advantage of these plans over linear plans is that they avoid the significant redundant computation inherent in executing all feasible linear plans separately. In our example, plan p_1 and p_2 will both execute source queries S_{11}^{fbf} and S_{31}^{bff} with the same binding patterns. In contrast, p_1' avoids this redundant access.[1]

Need for *searching* in the space of parallel plans: One remaining question is whether we should search in the space of parallel plans directly, or search in the space of linear plans and post-process the linear plans into a set of equivalent parallel plans. (An example of the post-processing approach may be one which generates top N plans using methods similar to those outlined in [DL99] and then parallelizes them). However such approaches in general are not guaranteed to give cost-coverage tradeoffs that are attainable by searching in the space of parallel plans because (i) The cost of generating a single best parallel plan can be significantly lower than the cost of enumerating, rank-ordering the top N linear plans and post-processing them and (ii) Since the post-processing approaches de-couple the cost and coverage considerations, the utility of the resulting plans can be arbitrarily far from the optimum.

Moreover, we will see that the main possible objection to searching in the space of parallel plans–that the space of

[1] Notice that here we are assuming that the linear plans are all executed independently of one another. A related issue is the optimal way to execute a union of linear plans–they can be executed in sequence, with cached intermediate results (which will avoid the redundant computation, but increases the total execution time), or execute them in parallel (which reduces the execution time while keeping the redundant accesses). These two options are really special cases of the more general option of post-processing the set of linear plans into a minimal set of parallel plans and executing them (see below).

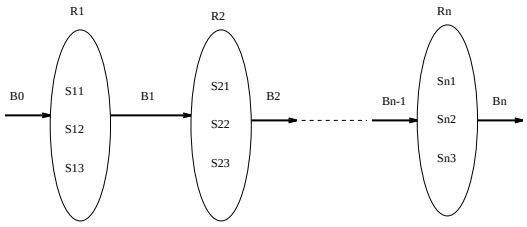

Figure 1: *Execution semantics of parallel plans*

parallel plans is much larger than the space of linear plans– turns out to be a red herring. Specifically, we will describe a way of searching in the space of subogoal orders, and for each subgoal order efficiently generating an optimal parallel plan. This approach winds up adding very little additional planning overhead over that of searching in the space of linear plans, and even this overhead is more than made up for by the fact that we avoid the inefficiencies of phased optimization. Consequently, in this paper, we consider joint optimization of cost and coverage in the space of parallel query plans.

3 Preliminaries

3.1 Schemas and queries

A *schema* consists of the collections and relations in an actual or virtual data set. A *query* is an expression indicating the data desired in terms of the schema. A *view* is simply a named query. In this paper we express queries and views as datalog rules.

Data integration systems provide their users a virtual *mediated schema* to query over. This schema is a uniform set of relations serving as the application's ontology and is used to provide the user with a uniform interface to a multitude of heterogeneous external data sources, which store the actual available data. We model the contents of these external data sources with a set of *source relations* which are defined as views over the mediated schema relations.

In data integration systems, the users' queries are based on the mediated schema. Let's assume $R_1(\overline{X}_1)$, $R_2(\overline{X}_2)$, ..., $R_n(\overline{X}_n)$ are mediated schema relations, a query in our data integration system has the form:

$Q(\overline{X})$:- $R_1(\overline{X}_1)$, $R_2(\overline{X}_2)$, ..., $R_n(\overline{X}_n)$

The atom $Q(\overline{X})$ is called the *head* of the datalog rule, and the atoms $R_1(\overline{X}_1)$, $R_2(\overline{X}_2)$, ..., $R_n(\overline{X}_n)$ are the *subgoals* in the *body* of the rule. The tuples $\overline{X}$, $\overline{X}_1$, $\overline{X}_2$, ..., $\overline{X}_n$ contain either variables or constants, and we need $\overline{X} \subseteq \overline{X}_1 \cup \overline{X}_2 \cup ... \cup \overline{X}_n$ for the query to be safe.

3.2 Parallel information gathering plans

A parallel information gathering plan p has the form

$$p = ((...(sp_1 \bowtie sp_2) \bowtie ...) \bowtie sp_{n-1}) \bowtie sp_n,$$
$$sp_i = (S_{i1} \cup S_{i2} \cup ... \cup S_{im_i})$$

where sp_i is a subplan and S_{ij} is a source relation corresponding to the i^{th} subgoal of a subgoal order of a query. The execution semantics of subplan sp_i are that it queries its sources in parallel and unions the results returned from the sources. The semantics of plan p are that it executes the subplans in the order of $sp_1 \rightarrow sp_2 \rightarrow ... \rightarrow sp_n$, and joins the results of subplans, and returns the results as the answer to the query.

To clarify this process more, we need the concept of *binding relations*[2], which are intermediate relations that keep track of the partial results of executing the first k subplans of the query plan. Given an information gathering plan of n subgoals in the order of R_1, R_2, ..., R_n, we define a corresponding sequence of $n+1$ *binding relations* B_0, B_1, B_2, ..., B_n (see Figure 1). B_0 has the set of variables bound in the query as its schema, and it has a single tuple, denoting the bindings specified in the query. The schema of B_1 is the union of the schema of B_0 and the schema of the mediated relation of R_1. Its instance is the join of B_0 and the union of the source relations in the subplan of R_1. Similarly we define B_2 in terms of B_1 and the mediated relation of R_2, and so on. The answer to the conjunctive query is defined by a projection operation on B_n.

Example: We noted that $p_1{}'$ is a possible parallel query plan for the example in Section 2:

$p_1^{'}$=$((S_{11}^{fbf} \cup S_{12}^{fbf}) \bowtie (S_{21}^{bf} \cup S_{22}^{bf})) \bowtie S_{31}^{bff}$,

The execution of $p_1^{'}$ involves the following steps:

1. Query the sources S_{11} and S_{12} in parallel by binding the attribute "title" as "Data Warehousing", then we union the results returned by these two sources.
2. Store the unioned results into a result relation (called binding relation) B_1.
3. Use the ISBN values of the binding relation to query sources $S_{21}$and S_{22} in parallel, and union the results from these two sources.
4. Join the binding relation B_1 with these new union results to get the new binding relation, B_2
5. Use the ISBN values in the binding relations to query source S_{31}.
6. Join the results with the binding relation B_2 to get B_3.

The tuples in B_3 constitute the answer to the original query.

4 Cost and Coverage Models

The main aim of this section is to describe the models we use to assess the execution cost and coverage of (parallel) query plans, and how these are combined to compute the utility of the plan. We start by describing the statistics we assume, and then describe how the execution cost and coverage of a plan are computed using these statistics. Before we go further, we must note that *the contributions of this paper are largely independent of the exact statistics assumed in this section, as well as the specific way in which we combine cost and coverage into utility*. Our intent is to give an idea

[2]The idea of binding relations is first introduced in [YLUG99] for linear information gathering plans where each subgoal of the query has only one source relation. We use a generalization of this idea to parallel plans.

of how cost and coverage of a plan can be computed and combined.

For a source S defined over the attributes $A = \{A_1, A_2, ..., A_m\}$ and the mediated schema defined in the data integration system as $R_1, R_2, ..., R_n$, we currently assume the following statistical data. These assumptions are in line with the types of statistics used by previous work (c.f. [LRO96,NLF99]), and techniques for "learning" some of these statistics through probing are available in the literature (c.f. [GRZ+00]).

1. For each attribute A_i, its length (in number of bytes), denoted by $length(A_i)$, and for each attribute A_i in source relation S, the number of distinct values of A_i, denoted by $card(\prod_{A_i}(S))$;

2. The number of tuples in source S, denoted by $card(S)$, and the feasible binding patterns of the source S denoted by $bps(S)$;

3. For each mediated relation R_j, its coverage in the source S , denoted by $P(S|R_j)$, for example, $P(S|author) = 0.8$ denotes that source S stores 80% information of the mediate relation $author(name, title)$ of all the sources in the data integration system. Following [NLF99,FKL97], we also make the simplifying assumption that the sources are "independent" in that the probability that a tuple is present in source S_1 is independent of the probability that the same tuple is present in S_2.

4. The local delay time for the source S to process a query with 1 kbyte answer data, denoted by $localDelay(S)$;

5. The average speed of transfer data from the data integration system to the source S, denoted by $bandWidth(S)$, and the average time of sending a message from the data integration site to the source site, denoted by $msgDelay(S)$;

We note that while some of these statistics may be provided by the individual sources, others might be learned from experience by the mediator. For example, in order to get the bandwidth statistics, the data integration system has to average the transfer rates of every query to the source. For the message delay statistics the data integration system will average the message response time of every message communication to source.

Estimating the Cost of a parallel plan:

In this paper, we will estimate the cost of a parallel plan purely in terms of its execution time. We will also assume that the execution time is dominated by the tuple transfer costs, and thus ignore the local processing costs at the mediator. Thus the execution costs are computed in terms of the response times offered by the various sources that make up the (parallel) plan. The response time of a source is proportional to the number of times that the source is called, and the expected number of tuples transferred over each call. Since the sources have binding pattern limitations, and the set of feasible source calls depend on the set of call variables that can be bound, both the number of calls and the number of tuples transferred depend on the value of the binding relation preceding the source call.

Specifically, suppose we have a plan p with the subplans $\{sp_1, sp_2, ..., sp_n\}$. The cost of p is given by:

$cost(p) \doteq \sum_i responseTime(sp_i)$

The response time of each subplan $sp_i (= \{S_{i_1}, S_{i_2}, ..., S_{i_m}\} \in p)$ is computed in terms of the response times of the individual sources that make up the subplan. Since the sources are processed in parallel, the cumulative response time is computed in terms of the maximum response time of all the sources in the subplan:

$responseTime(sp_i) =$
$\quad max_{j \in [1,m]}\{responseTime(S_{i_j}, B_{i-1})\}$

Notice that the response time of each source is being computed in the context of the binding relation preceding that source call. We will now describe how we estimate of the response time of a source under a given binding relation. We start by noting that the different sources will have different querying capabilities, with some sources supporting parallel processing of several queries, and others having significant binding restrictions on the allowed source calls.

In order to estimate the least costly way to access the source, we have to take into account the source querying capabilities, and the available binding relation.
Let us assume that the source relation is $S(Y_1, Y_2, ..., Y_m)$, and the binding relation preceding the source call is $B(X_1, X_2, ..., X_n)$. We then have

$responseTime(S, B) =$
$\quad msgDelay(S) * msgs(S, B) +$
$\quad bytes(S, B) * (localDelay(S) + bandWidth(S))$

where $msgs(S, B)$ is the number of separate calls made to the source S under the binding relation B, and $bytes(S, B)$ denote the total bytes sent back by the source S in response to these calls. The rest of the expressions are as described in the source statistics that we assume; see the beginning of this section. Both $msgs(S, B)$ and $bytes(S, B)$ depend on the specific strategy used to query the source S.

Basically there are two broad strategies, one is to get S to *ship the whole relation*, and the other is to respect the querying restrictions and tuple transfer costs on S and *fetch the relation as needed*. Since most internet sources tend to preclude requests for shipping the whole relation, we are often left with the second strategy. The basic idea here is similar to the use of "semi joins" in distributed databases [OV99], where only the join attribute of binding relation B is shipped to source S. S then performs a join with this single attribute, and all the joining tuples are sent back to the mediator (where the actual join between B and S is done). One complicating factor is that unlike traditional databases, many sources on the Internet have limitations on the number of tuples included in a single query (For example, most stock quote sources typically limit the number of securities whose quotes are requested in a single call to under 10). This limitation in the disjunctive query capability precludes sending multiple query tuples at the same time. If the source has a disjunction limit of d (in that it can accept at most d query tuples at the same time), then we need to split the binding attribute tuples being sent to S into packets of size d. This increases the number of separate calls made to the source, and consequently the response time. A final twist is

that sometimes the sources do allow multiple parallel calls to be made. If a source allows p parallel calls to be made, then the response time can be reduced by packaging the total number of source accesses into call sets of size p. Putting this all together, we have the following formula for computing the best-case response time of a source S which has a disjunction limit of d and can support up to p parallel accesses together:

$$responseTime(S,B) = msgDelay(S) * \left\lceil \frac{\left\lceil \frac{\prod_1^k(card(\Pi_{X_i}(B)))}{d} \right\rceil}{p} \right\rceil + \left\lceil \prod_1^k(card(\Pi_{X_i}(B)))/p \right\rceil * \frac{card(S)}{\prod_1^k(card(\Pi_{Y_i}(S)))} * (\sum_{i=1}^m length(Y_i)) * (localDelay(S) + bandWidth(S))$$

Here, k is the number of common (join) attributes between the binding relation B and the source relation S. Notice that we use $P(S|R)$ as part of the expression for estimating the number of tuples transferred by the source S for each call. Since all the factors used in this expression are part of the statistics that we assume in our model, computing the response time of a source under a given binding relation is straightforward.

Estimating the Coverage of a parallel plan: For a plan $p = \{sp_1, sp_2, ..., sp_n\}$, the coverage of p will depend on the coverages of the subplans sp_i in the p and the join selectivity factor of the subgoals associated with these subplans. We use $SF_J(R_{sp_{i_1}}, R_{sp_{i_2}})$, $1 \leq i_1 < i_2 \leq n$, to denote the join selectivity factors of any join combination of two subgoal relations of the subplans in the plan.

$$coverage(p) = \prod_{i=1}^n [P(sp_i|R_{sp_i})] \times \prod_{i_1<i_2} [SF_J(R_{sp_{i_1}}, R_{sp_{i_2}})]$$

Since we just want to compare the plans for different subgoal orders of the subgoals in the same subset, the expression $\prod_{i_1<i_2} [SF_J(R_{sp_{i_1}}, R_{sp_{i_2}})]$ remains constant. Comparison can thus be done just in terms of $\prod_{i=1}^n [P(sp_i|R_{sp_i})]$.

For each subplan sp_i in the plan p we compute how much information we can get by executing the source calls in this subplan. Let R_{sp_i} be the subgoal of the subplan sp_i=$\{S_{i_1}, S_{i_2}, ..., S_{i_m}\}$. We have:

$$\begin{aligned} P(sp_i|R_{sp_i}) &= P(\cup_{S_{i_j} \in sp_i} S_{i_j}|R_{sp_i}) \\ &= P(S_{i_1}|R_{sp_i}) + P(S_{i_2} \wedge \neg S_{i_1}|R_{sp_i}) + ... \\ &\quad + P(S_{i_m} \wedge \neg S_{i_1} \wedge ... \wedge \neg S_{i_{m-1}}|R_{sp_i}) \end{aligned}$$

As mentioned earlier, we assume that the contents of the sources are independent of each other. That is, the presence of a tuple in one source does not change the probability that the tuples also belongs to another source. Thus, the conjunctive probabilities can all be computed in terms of products. E.g.

$$P(S_{i_2} \wedge \neg S_{i_1}|R_{sp_i}) = P(S_{i_2}|R_{sp_i}) * (1 - P(S_{i_1}|R_{sp_i}))$$

4.1 Combining Cost and Coverage

In order to find the best plan, which can produce highest quality answers for a user's query with the cheapest cost, a data integration system has to evaluate the plans based on a utility function that combines the cost and coverage together.

The main difficulty in combining the cost and the coverage of a plan into a utility measure is that, as the length of a plan (in terms of the number of subgoals covered) increases, the cost of the plan increases additively, while the coverage of the plan decreases multiplicatively. In order to make these parameters combine well, we take the sum of the logarithm of the coverage component and the negative of the cost component.[3] The logarithm ensures that the coverage contribution of a set of subgoals to the utility factor will be additive.

$$utility(p) = w * log(coverage(p)) - (1 - w) * cost(p)$$

The user can vary w from 0 to 1 to change the relative weightage given to cost and coverage.[4] An advantage of this combination approach is that we can compute the utility of all subplans of a plan separately, and add them together as the utility of the whole plan.

5 Conditions for the existence of single parallel plans of maximal utility

In Section 6, we will develop algorithms for finding the single best parallel plan for any given query. While such a best single parallel plan will often have a significantly better utility (lower cost, higher coverage) than the best single linear plan (see the example in Section 2), it may not provide the maximal possible utility. In general, to achieve maximal possible utility, we may need to execute a union of parallel plans (note that executing additional plans increases the cost component as well as the coverage component of the utility– thus the improvement in utility will have to come from increased coverage). In practice, this may not be a major issue. To begin with, as the results in Section 7 will show the best single parallel plan often gives a utility that is more than 80% of the maximal attainable utility. Even in cases additional utility (coverage) is desired, we can either extend the algorithms themselves so they will iteratively output ranked list of plans until the user is satisfied, or can extend the best parallel plan at execution time (by union-merging binding-compatible sources into the plan) to provide additional utility.

It is however of theoretical interest to understand the conditions under which the maximal utility can be attained with a single parallel plan. This is what we try to do in the current section. Whether or not a single parallel plan with highest possible utility exists depends on the aggregate binding pattern restrictions of the sources. To formalize this, we define

[3] We adapt this idea from [C01] for combining the cost and quality of Multimedia database query plans, while the cost is also increases additively and the quality (such as precision and recall) decreases multiplicatively when the number of predicate increases.

[4] In the actual implementation we scale the coverage appropriately to handle the discontinuity at 0, and to make the contribution from the coverage component to be in the same range as that from the cost component.

the notion of binding restrictions and preferred binding patterns for query subgoals in terms of the corresponding restrictions and preferences for the relevant sources for those subgoals.

Let $Bucket(R)$ be the set of sources that are relevant to the subgoal R. The *binding pattern restriction* of a subgoal is defined as the set of most general binding patterns that still satisfy the restrictions of all the sources relevant to that subgoal. For example if a subgoal $R(X_1, X_2, X_3)$ has three sources $S_1(X_1^f, X_2^f, X_3^f)$, $S_2(X_1^{b/f}, X_2^{f/b}, X_3^f)$ (X_1 or X_2 must bind), $S_3(X_1^f, X_2^b, X_3^f)$, in its buckets, the binding pattern restriction will be $R(X_1^f, X_2^b, X_3^f)$.

Theorem 1 *If there is a subgoal order which can resolve all the binding pattern restrictions of the subgoals in the query, and the utility is defined entirely in terms of coverage, then there is an optimal single parallel plan of highest possible utility that will cover all the answers of the query.*

Since at least one subgoal order exists that resolves all the binding restrictions, we can construct a parallel plan that has the subgoals in that order, and the subplan for each subgoal consists of *all* the sources in the bucket of that subgoal. Since utility is defined fully in terms of coverage, this single parallel plan will have the maximum possible utility.

We will now try to characterize the sufficient conditions for the case when the utility is defined both in terms of cost and coverage. Although we can query a source using any of its feasible binding patterns, the execution costs will depend on the exact binding patterns used in accessing the sources (see Section 4), which in turn depend on the binding relation that holds before that subgoal. The *preferred binding pattern of a source S under a given binding relation B* is the feasible binding pattern p of the source S such that accessing S with p will lead to the lowest execution cost under B. The preferred binding pattern of a subgoal R under a binding relation B is defined as the binding pattern p that is the preferred binding pattern of a majority of the sources in the bucket of R.

Theorem 2 *If (i) For every possible binding relation B the preferred binding pattern p of each subgoal R under B is the same as the preferred binding pattern of each of the sources $S \in Bucket(R)$ under B and (ii) there is at least one subgoal order which can resolve all the binding pattern restrictions of the subgoals in the query, then there is an optimal single parallel plan that will maximize the utility function, even if it is defined in terms of both coverage and cost.*

Although the condition (i) in the theorem above seems too restrictive at first glance, it does in fact hold in fairly general situations. For example, it holds in the following two cases:

1. All the sources in the bucket of a subgoal have cost models that satisfy "bound is easier" assumption. That is, given a source $S(x_1, x_2)$, and a binding relation B preceding it, the preferred binding pattern of S under B will be the feasible binding pattern of S that binds all the variables for which values are supplied by B (if B provides values for x_1 and not x_2, then the preferred binding pattern of S will be $[b, f]$).
2. All the sources in the bucket of a subgoal have cost models that satisfy the "least accesses is best" assumption, which means that it is best to access the source with the most general feasible binding pattern given the binding relation B and the binding restrictions of S. For example, suppose we have the source $S(x_1, x_2)$, such that all feasible binding patterns of S require x_2 to be bound. Suppose the binding relation B provides bindings for both x_1 and x_2. Under this assumption, the least costly way to access S would be to use the binding pattern $[f, b]$ (and do selection over x_1 at the mediator site).

Both the "bound is easier" assumption and the "least accesses is best" assumptions have been used as the basis for greedy algorithms for query optimization in information integration (c.f. [LRO96;KG99]).

6 Generating query plans

Now that we have described the details of computing the utility of a plan, we are ready to present algorithms for query planning. The algorithms presented in this section aim to find a single best parallel plan–i.e., the parallel plan with the highest utility. As Theorems 1 and 2 in Section 5 show, such a plan is not guaranteed to have the highest achievable utility in all cases. Nevertheless, as our empirical results shall demonstrate, not only is it significantly cheaper to generate the best single parallel plan than it is to enumerate, rank and execute many linear plans; executing the best single parallel plan typically also offers a significantly higher utility than executing several linear plans.

Our basic plan of attack involves considering different feasible subgoal orderings of the query, and for each ordering, generating a parallel plan that has the highest utility. To this end, we first consider the issue of generating the best plan for a given subgoal ordering.

Given the semantics of parallel plans (see Figure 1), this involves finding the best "subplan" for a subgoal relation under a given binding relation. We provide an algorithm for doing this in Section 6.1. We then tackle the question of searching the space of subgoal orders. For this, we first provide a dynamic programming algorithm (Section 6.2) and then a greedy plan generator (Section 6.3).

6.1 Subplan Generation

For a subgoal R with m sources $S_1, S_2, ..., S_m$ in its corresponding bucket, we provide a CreateSubplan algorithm (see Algorithm 1) to compute the best subplan for the subgoal under the binding relation B.

The algorithm first computes the utility of all the sources in the bucket, and then sorts the sources according to their utility value. Next the algorithm adds the sources from the sorted bucket to the subplan one by one, until the utility of the current subplan is less than the utility of the previous subplan. We use the models discussed in Section 4 to calculate the cost and coverage of the sources.

Although the algorithm has a greedy flavor, the subplans generated by this algorithm can be shown to be optimal if

Algorithm 1 CreateSubplan

```
1: input: B: the binding relation; R : the subgoal in the query
2: output: sp : the best plan
3: begin
4: sp ← {}
5: Bucket ← the Bucket for the subgoal R;
6: for each source s ∈ Bucket do
7:    if (s is feasible under B) then
8:       utility(s) = w * log(coverage(s)) − (1 − w)responseTime(s, B);
9:    else
10:      remove s from Bucket
11:   end if
12: end for
13: sort the sources in Bucket in decreasing order of their utility(s);
14: s ← the first source in the sorted Bucket;
15: while ( s != null) and (utility(sp + {s}) > utility(sp) ) do
16:   sp ← sp + {s}
17:   s ← the next source in the Bucket;
18: end while
19: return sp;
20: end
```

the sources are conditionally independent [FKL97] (i.e., the presence of an object in one source does not change the probability that the object belongs to another source). Under this assumption, the order of the sources according to their coverage and cost will not change after we execute some selected sources.

The running time of the algorithm is dominated by line 15, which is executed m times, taking $O(m)$ time in each loop for computing the utility of the subplan (under the source independence assumption). Thus the algorithm has $O(m^2)$ complexity.

6.2 A Dynamic Programming Approach for Parallel Plan Generation

In the following we introduce a dynamic programming-style algorithm called *ParPlan-DP* which extends the traditional System-R style optimization algorithm to find the best parallel plan for the query. The basic idea is to generate the various permutations of the subgoals, compute the best parallel plan (in terms of utility) for each permutation, and select the best among these. Although the outlines are similar to a traditional system-R style algorithm and [FLMS99], there are some important differences:

1. Our algorithm does source selection and subgoal ordering together according to our utility model for parallel plans; while the traditional System-R and [FLMS99] just needs to pick a best subgoal order according to the cost model.

2. In order to generate a parallel subplan, the algorithm uses the CreateSubplan algorithm to general subplan for each subgoal of each subgoal order, in contrast the traditional System-R and [FLMS99], which need only find the best access strategy for a single source (relation) of a subgoal.

3. We also have to estimate attribute sizes of the binding relations for partial parallel plans, where there are multiple sources for a single subgoal. So we have to take the overlap of sources in the subplan into account to estimate the sizes of each attributes in the binding relation. For a given subgoal R and the binding relation B_- that holds before R in the plan, assume the best subplan for R is $sp = \{S_1, S_2, ..., S_m\}$. Then we can estimate the sizes of the binding attributes in binding relation B_+ after querying all the sources in the subplan using the binding relation estimation formula. Let's assume the join attributes of relation B_- and S are $X_i = Y_i, i = 1, 2, ..., k; 1 \leq k \leq min(m, n)$, and that X_j are the attributes that are in both S and B_+, but not in B_-).

$B_+ = B_- \bowtie (\bigcup_{i=1}^{m} S_i)$ where $1 \leq i \leq m$

$card(\Pi_{X_j}(B_+)) = (card(\Pi_{X_j}(B_- \bowtie S_1))) +$

$\frac{P(S_2 \wedge \neg S_1 | R)}{P(S_2|R)} * (card(\Pi_{X_j}(B_- \bowtie S_2))) + ... +$

$\frac{P(S_m \wedge \neg S_1 \wedge ... \wedge \neg S_{m-1} | R)}{P(S_m|R)} * (card(\Pi_{X_j}(B_- \bowtie S_m)))$

Algorithm 2 *ParPlan-DP*

```
1: Input: BUCKETS : Buckets for the n subgoals
2: output: p : the best plan
3: begin
4: S← {}; {a queue to store plans;}
5: p0.plan← {}; {p0: the initial node}
6: p0.B← B0; {the binding relation of p0: B0}
7: p0.R← {}; {the selected subgoals of p0: empty}
8: p0.utility← −∞; {the utility of p0: negative infinity}
9: S← S + {p0};
10: p← pop the first plan from S;
11: while (p ≠ null) and (# of subgoals p.R < n) do
12:    for each feasible subgoal Ri(∈ BUCKETS and ∉ p.R) do
13:       make a new plan p' ;
14:       sp ← CreateSubplan(p.B, Ri);
15:       p'.plan ← p.plan + sp;
16:       m ← # of sources in sp;
17:       p'.B←p.B ⋈ (⋃_{i=1}^{m} Si); {Si ∈ sp}
18:       p'.R← p.R + {Ri};
19:       p'.utility←utility(p');
20:       if ( ∃p1 ∈ S ) and (p1.R commutatively equals p'.R) and
          (p'.utility > p1.utility) then
21:          remove p1 from S and push p' into S
22:       else if (p'.utility ≤ p1.utility) then
23:          ignore p'
24:       else
25:          push p' into S;
26:       end if
27:    end for
28:    p← pop the first plan from S;
29: end while
30: return p;
31: end
```

Algorithm 2 finds the optimal plan by enumerating all possible subgoal orders restricted to left-deep trees. Suppose we want to find the best plan for the query with R_1, R_2, ..., R_n as its subgoals. In our algorithm, we use a queue data structure to remember all the best partial plans for every subset of one or more of the n subgoals. For each subset, we store the following information: (i) the best plan for these subgoals, which include optimal source selection for each subplan of the subgoals in the subset , the subgoal execution order and the best binding patterns for querying these selected sources; (ii) the binding relation and binding attributes after querying all the sources of the best plan; (iii)

the utility of the best plan; and (iv) the coverage of the best plan (used to compute the coverage for any extensions of this best plan). In contrast, a traditional system-R style optimizer need only track the best plan, and its cost [SACL79].

The subgoal permutations are produced by the dynamic construction of a tree of alternative plans. First, the best plan for any one subgoals is considered, followed by the plan of two subgoals. This continues until the best plan for n subgoals is found. When we have the best plan for any i subgoals, we can find the best plan for $i+1$ subgoals by using the results of first i subgoals and finding the best subplan for the $i+1$ th subgoal under the binding relation given by the subplans of the first i subgoals. Actually, the algorithm does not generate all possible permutations since some of them are useless. Permutations involving subgoals without any feasible source queries are eliminated, as are the commutatively equivalent permutations with the lowest plan utility.

6.2.1 Complexity Analysis

The worst case complexity of query planning with *ParPlan-Greedy* is is $O(2^n m^2)$, where n is the number of subgoals in the query and m is the number of sources exporting each subgoal. The 2^n factor comes from the complexity of traditional dynamic programming, and the m^2 factor comes from the complexity of CreateSubplan. The fact that the complexity is exponential in terms of the subgoal number n, but only polynomial in terms of number of sources per subgoal is good. This is because in a typical data integration scenario, the number of subgoals in a query may be much smaller than the number of sources exporting each subgoal. Finally, it is instructive to note that the complexity of our approach is significantly less than the full size of the space of parallel plans, which is[5] $O(n!2^{m^n})$. This happens mainly because we are using a greedy way of generating subplans for individual subgoal orders.

We also note that searching in the space of parallel plans has *not significantly* increased the complexity of our query planning algorithm. In fact, our $O(2^n m^2)$ complexity compares very favorably to the complexity of the linear plan enumeration approach described in [LRO96], which will be $O(m^n n^2)$, where m^n is the number of linear plans that can be enumerated, and n^2 is the complexity of the greedy algorithm they use to find the feasible execution order for each linear plan. This is despite the fact that the approach in [LRO96] is only computing *feasible* rather than optimal execution orders (the complexity would be $O(m^n 2^n)$ if they were computing optimal orders).

6.3 A Greedy Approach

We noted that *ParPlan-DP* is already significantly faster than the linear plan enumeration approaches. Nevertheless, it is exponential in the number of query subgoals. In order to get a more efficient algorithm, we need to trade the optimality guarantees for performance. We introduce a greedy algorithm *ParPlan-Greedy* (see Algorithm 3) which gets a plan quickly at the expense of optimality. This algorithm gets a feasible execution plan by greedily choosing the subgoal which can give the maximum plan utility from the unchosen subgoals as the next subgoal to achieve in the plan. The inputs are the buckets for all the subgoals in the query.

Algorithm 3 Greedy

```
1:  Input: BUCKETS : Buckets with n subgoals; p_0 : initial plan;
2:  UCS_0: unchosen subgoals; B_0 : initial binding relation;
3:  output: p : the best plan
4:  begin
5:  B← B_0
6:  UCS← UCS_0;
7:  p← p_0
8:  while (UCS ≠ {}) do
9:     for each feasible subgoal R_i(∈ UCS ) do
10:       sp_i ← CreateSubplan(B, R_i)
11:    end for
12:    sp_max ←subplan which will maximize {utility(p + sp_i)}
13:    p← p + sp_max; {assume the plan p + sp_max has the maximum
       utility}
14:    UCS← UCS − {R_max}
15:    m ← # of sources in sp_max;
16:    B←B ⋈ (⋃_{i=1}^{m} S_i); {S_i ∈ sp_max}
17: end while
18: return p;
19: end
```

We set the input P_0 ={}, UCS_0={*all the subgoals in the query* Q} and B_0={*the bindings specified in the query* Q}. The worst-case running time of *ParPlan-Greedy* is $O(n^2 m^2)$, where n is the number of subgoals in the query, and m is the number of sources per subgoal.

Theoretically, *ParPlan-Greedy* may produce plans that are arbitrarily far from the true optimum, but we shall see in Section 7 that its performance is quite fair in practice. It is of course possible to use this *ParPlan-Greedy* in concert with a hill-climbing(algorithm 4) approach to iteratively improve the utility.

The output of the greedy best-first algorithm is used as the input to the Hill-Climbing algorithm. The individual iterations of the Hill-Climbing algorithm may in turn involve use of the Best-First algorithm to get another modified plan. Since this is an iterative improvement algorithm, we can terminate it within a prescribed time limit (iteration limit).

Since the worst-case running time of the Best-First algorithm is n^2, it is also the individual iteration time of the hill-climbing procedure. We can control the total running time of the Hill-Climbing algorithm in terms of TL.

7 Experiments

We implemented the query planning algorithms described in this paper on a Solaris machine using JDK1.2.2. In this section, we describe the results of a set of experiments we conducted with these algorithms. The goals of our experiments are:

- To compare the running time and quality of the solutions provided by our algorithms with the approaches that enumerate, rank and execute linear plans. For this

[5] For each subgoal there can be 2^m possible subplans–corresponding to the subsets of its bucket, and since there are n subgoals, there are 2^{m^n} ways of selecting subplans for the individual subgoals. Each selection of subplans can be arranged in $n!$ permutations

Algorithm 4 HillClimbing

Input: $BUCKETS$: Buckets for the n subgoals, p_0:initial plan, TL: time limit
Output: p : result plan

begin
$time \leftarrow 0$;
$p \leftarrow p_0$;
while (time<TL) **do**
  $p' \leftarrow p$;
  randomly pick a subplan sp in p';
  delete sp and all the subplans followed by sp from p';
  randomly pick a subgoal R in $BUCKETS$ such that its corresponding subplan is not in p';
  $p' \leftarrow p$ + CreateSubplan(binding relation of p', R);
  B$\leftarrow$binding relation of p';
  UCS$\leftarrow$subgoals in $BUCKETS$ but their corresponding subplans are not in p';
  $p' \leftarrow$BestFirst($BUCKETS$,p',UCS, B);
  if (utility(p')>utility(p)) **then**
    $p \leftarrow p'$;
  end if
end while
return p;
end

purpose, we implemented the algorithms described in [LRO96], which enumerates all linear plans, finds feasible execution orders for all of them, and executes them.

- To demonstrate that our algorithms area capable of handling a spectrum of desired cost-quality tradeoffs.

- To compare the performance of *ParPlan-DP* and *ParPlan-Greedy*.

Our experiments were run on a SUN ULTRA 5 with 256Mb of Memory. Our current experiments were done with a set of simulated sources. We designed 206 artificial data sources and 10 mediated relations covering all these sources. The statistics for these sources were generated randomly, 60% sources have coverage of $20\% - 40\%$ of their related mediated relations, 20% sources have coverage of $40\% - 80\%$, 10% sources have coverage below 20%, and 10% sources have coverage above 80%. 90% of the sources have binding pattern limitations. We also set the response time statistics of these sources randomly: 20% of sources have high response time, 20% of them have low response time, and 60% of them have the medium response time.

Variation of planning cost w.r.t. number of subgoals: Figure 2 compares the planning time for our algorithms with the approach in [LRO96], as the query size is increased from 1 to 10, while the number of sources per subgoal is kept constant at 8. The planning time for [LRO96] consists of the time taken to produce all the linear plans and find a feasible execution order for each plan using the greedy approach in [LRO96], while the time for our algorithms consists of the time taken to construct and return the first parallel plan (which is also the best plan in the case of the *ParPlan-DP*).

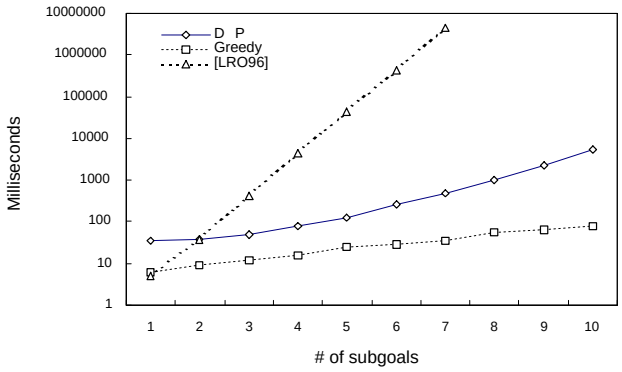

Figure 2: *Variation of running time with the query size (when the the number of relevant sources per subgoal is held constant at 8). X axis plots the query size while Y axis plots the running time.*

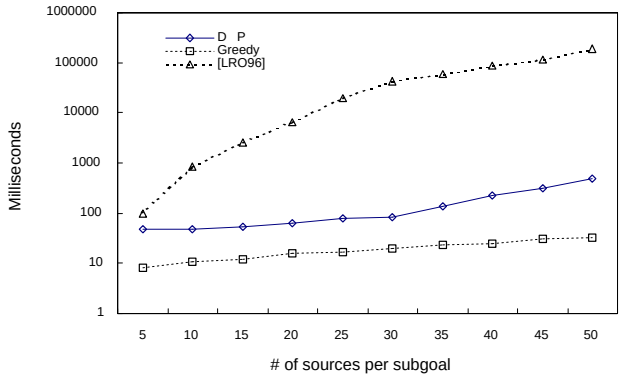

Figure 3: *Variation of running time with number of relevant sources per subgoal (for a query of size 3). X axis plots the query size while Y axis plots the running time.*

We see right away that both our algorithms incur significantly lower planning time than the decoupled approach used in [LRO96]. We also note, as expected, that *ParPlan-Greedy* scales much better than the exhaustive one.

Variation of planning cost w.r.t. number of sources: Figure 3 studies the effect of varying the number of sources exporting each subgoal. Scalability with respect to the number of sources per subgoal is, in a sense, more critical, since it is reasonable to expect that normal queries posed by users will not have too many conjuncts in them, but harder to limit the number of sources exporting a query. We keep the number of subgoals constant at 3, and vary the number of sources per subgoal from 5 to 50. As we can see, the planning time for both of our algorithms grows almost linearly as the number of sources for each subgoal increases, while the planning time for [LRO96]'s enumeration degrades considerably.

Quality comparison: In Figure 4, we compare the quality of plans generated by *ParPlan-DP* and the enumeration algorithm given in [LRO96] for queries with 4 subgoals and each subgoal with 8 sources. The x-axis shows the weights used in the utility metric–with the relative weightage for coverage increasing from left to right. Notice that the algorithms

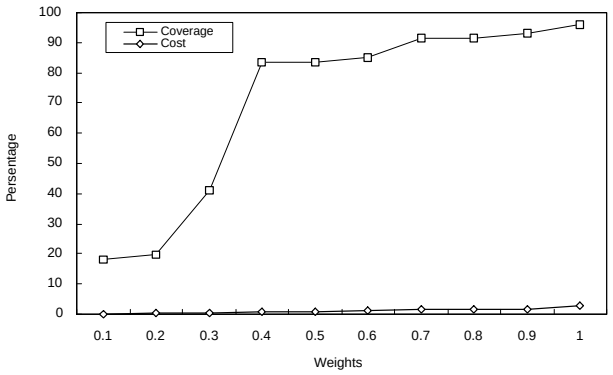

Figure 4: *Comparing the quality of the plans generated by ParPlan-DP algorithm with those generated by [LRO96] (for queries of 4 subgoals), while the weight in the utility measure is varied. X axis shows the weight value in the utility measure and Y axis plots the cost and coverage of our algorithm expressed as a percentage of the cost and coverage provided by [LRO96].*

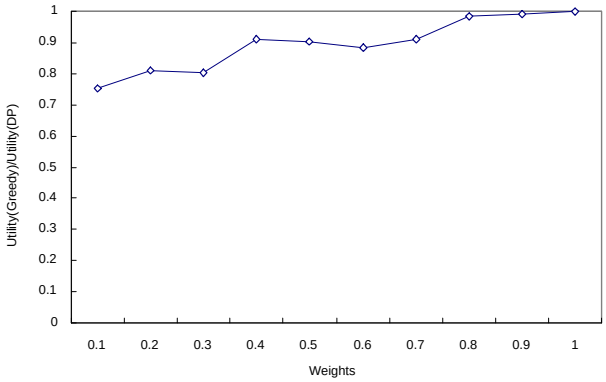

Figure 5: *Ratio of the utility of the plans given by ParPlan-Greedy to that given by ParPlan-DP for a spectrum of weights in the utility metric. X axis varies the weight used in the utility metric, and Y axis shows the ratio of utilities*

in [LR096] do not take account of the relative weighting between cost and coverage. So the cost and coverage of the plans produced by this algorithm is the same for all values of w. Thus, on the y-axis, we plot both the execution cost and coverage of *ParPlan-DP* as a percentage of the cost and coverage provided by [LRO96]. We notice that the best plan returned by our algorithm gives a pretty high coverage (over 80% of the coverage for w over 0.4) while incurring cost that is below 2% of that incurred by [LRO96]. Note also that even though our algorithm seems to offer only 20% of the coverage offered by [LRO96] at w=0.1, this makes sense given that at w=0.1, the user is giving 9 times more weight to cost than coverage (and the approach of [LRO96] is basically ignoring this user preference and attempting full coverage).

Comparing the greedy and exhaustive approaches: We also investigated the question: "exactly how bad are the plans produced by *ParPlan-Greedy* compared to that produced by *ParPlan-DP*?" A moment's reflection makes it clear that the answer depends on the specific weights used in the utility function. As an extreme example, if the utility function is concerned only about the coverage and not the cost, then the optimal plans will involve calling all the relevant sources (if their binding patterns allow the call). Both the algorithms should be able to produce such a plan easily, and thus *ParPlan-Greedy* should be as good as *ParPlan-DP*. The relative quality is likely to diverge considerably as the utility function starts taking cost into account and the subgoal binding pattern performance diverges. In order to see the way the relative performance varies, we experimented with queries with 4 subgoals and each subgoal with 8 sources, while the utility function is varied from being biased towards cost to being biased towards coverage. Figure 5 shows the utility of the plan produced by *ParPlan-Greedy* as a fraction of the utility of the plan produced by *ParPlan-DP*. We observe, as expected, that the utility of plans given by *ParPlan-DP* is better than that of the *ParPlan-Greedy* with small initial weight (corresponding to a bias towards cost), with the ratio tending to 1 for larger weights (corresponding to a bias towards coverage). It is also interesting to note that at least in these experiments, the greedy algorithm is always producing plans that are within 70% of the utility of those produced by *ParPlan-DP*.

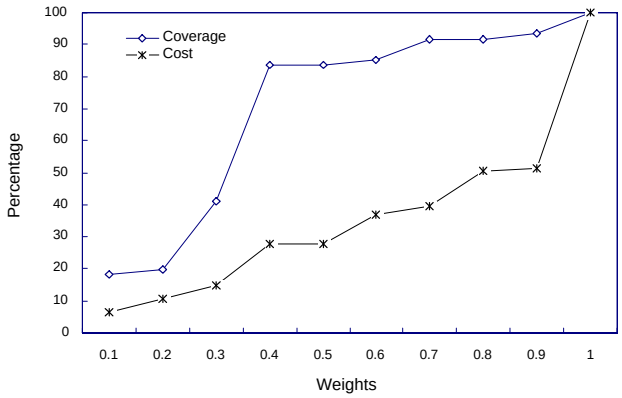

Figure 6: *Comparing the Coverage and cost of the plans found by ParPlan-DP by using different weights in Utility function, on queries of 4 subgoals. X axis varies the weights in the utility function, while the Y axis shows the cost and coverage as a percentage of the cost and coverage offered by [LRO96].*

Ability to Handle a spectrum of cost-coverage tradeoffs: Our final set of experiments was designed to showcase the ability of our algorithms to handle a variety of utility functions and generate plans optimized for cost, coverage or a combination of both. Figure 6 shows how the coverage and the cost of plans given by our *ParPlan-DP* changes when the weight of the coverage in the utility function is increased. The x-axis shows the weight of the coverage component. The y-axis shows the values of coverage and cost normalized in terms of the maximum coverage and maximum cost of the plans given by *ParPlan-DP*. We observe that as expected both the coverage and the cost increase when we try to get higher coverage. We can also see that for the particular query at hand, there is a large area in which the cost increases slowly while the coverage increases more rapidly.

An intriguing possibility offered by plots like this is that if they are done for representative queries in the domain, the results can be used to suggest the best initial weightings–those that are likely to give high coverage plans with relatively low cost–to the user.

8 Related work

Query processing in information integration context has received wide-spread attention of late. Early work in this area concentrated on modeling the information integration task. The approach adapted in our work is the so-called "local as view" approach that models the sources as views on the virtual global schema [DG97a;LRO96]. Bucket algorithm [LRO96] and the source inversion algorithm [DG97;DL97] are two approaches for generating candidate query plans in the local as view approach. As we saw in Section 2, we have adapted the bucket algorithm for our work. Once we know how to generate the candidate plans, the next issue is one of generating and executing a optimal query plan. This task is complicated by several issues. The sources may have partial overlap, and in order to avoid unnecessary calls to redundant information sources, we need to model the overlap between sources. Early approaches to this involved using the "local closed world assumptions" which attempt to capture the complete subsumption of one source by another over a particular query [LRO96;LK98]. A more sophisticated approach involves modeling the statistical overlap between sources, as is done the coverage computation used in this paper. The model and assumptions used in this paper are derived from the work described in [FKL97]. Another issue is modeling and respecting the limited access capabilities of the typical sources on the Internet. We do this in terms of binding patterns [LRO96] and the amount of disjunction allowed [YLUG99].

The work on Streamer project [DL99] extends the query planning algorithm in [LRO96], using the source overlap models in [FKL97] so it uses the coverage information to decide the order in which the potential plans are executed. A recent extension of [LRO96] is the MINICON algorithm presented in [PL00]. Although MINICON improves the efficiency of the bucket algorithm, it still assumes a decoupled strategy–concentrating on enumerating linear plans first, assessing their quality and executing them in a rank-ordered fashion next.

The work by Naumann et. al. [NLF99] offers another variation on the bucket algorithm of [LRO96], where the set of linear plans are ranked according to a set of quality criteria, and a branch and bound approach is used to develop top-N best linear plans. All these approaches use a "phased" optimization strategy, concentrating solely on the coverage and quality of plans during query planning. As we discussed in Section 2, this type of phased optimization can lead to significantly costly plan generation phase, as well as high plan execution costs.

Although [YLUG99] and [FLMS99] consider the cost-based query optimization problem in the presence of binding patterns, they do not consider the source selection issue in their work. Although the issues involved in source selection and ordering, as well as the need to combine them were discussed in [VP98], no specific algorithms are presented in that paper. Finally, the work on the GARLIC system [HKWY97] is similar in spirit to ours in that it too advocates a statistics based approach for query planning in the context of heterogeneous database integration. An important difference is that GARLIC does not deal with partially overlapping sources, which our algorithms explicitly do.

9 Conclusion

In this paper we started by motivating the need for joint optimization of cost and coverage of query plans in information integration. We then argued that our way of searching in the space of parallel query plans, using cost models that combine execution cost and the coverage of the candidate plans, provides a promising approach. We described ways in which cost and coverage of a parallel query plan can be estimated, and combined into an aggregate utility measure. We then presented two algorithms to generate parallel query plans. The first, *ParPlan-DP*, is a System-R style dynamic programming algorithm, while the second, *ParPlan-Greedy*, is a greedy algorithm. Our experimental evaluation of these algorithms demonstrates that for a given coverage requirement, the plans generated by our approach are significantly better, both in terms of planning cost, and in terms of the quality of the plan produced (measured in terms of its coverage and execution cost), compared to the existing approaches that use phased optimization using linear plans. Specifically, we showed that the query plans returned by our algorithm give over 80% of the coverage given by the exhaustive enumeration approach in [LRO96], while incurring only 2% of the execution cost incurred by the latter. We also demonstrated the flexibility of our algorithms in handling a spectrum of cost-coverage tradeoffs.

References

[ACPS96] S. Adali, K. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proceedings of SIGMOD-96*, 1996.

[AHK96] Yigal Arens, Chung-Nan Hsu, and Craig A. Knoblock. Query processing in the SIMS information mediator. In Austin Tate, editor, *Advanced Planning Technology*, pages 61-69. AAAI Press, Menlo Park, California, 1996.

[BRV98] L. Bright, L. Raschid, M. Vidal. Optimization of Wrappers and Mediators for Web Accessible Data Sources (WebSources). In *CIKM'98 Workshop on Web Information and Data Management* (WIDM'98),1998.

[C01] K.S. Candan. Query optimization in Multi-media and Web Databases. ASU CSE TR 01-003. Computer Science & Engg. Arizona State University.

[CGHI94] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantino, J.Ullman, J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *IPSJ*, Japan, 1994.

[CKPS95] S. Chaudhuri, R. Krishnamurthy, S. Potaminos and K. Shim. Optimizing queries with materialized views. *ICDE*. 1995.

[DL99] A. Doan and A. Levy. Efficiently Ordering Plans for Data Integration. The *IJCAI-99 Workshop on Intelligent Information Integration*, Stockholm, Sweden, 1999.

[DG97a] O.M. Duschka and M.R. Genesereth. Answering recursive queries using views. *In Proc. PODS*,1997.

[DG97b] O.M. Duschka and M.R. Genesereth. Query planning in infomaster. In *12th ACM Symposium on Applied Computing*, San Jose, CA, 1997.

[DL97] O. M. Duschka and A. Y. Levy. Recursive plans for information gathering. In *Proc. IJCAI*, 1997

[FKL97] D. Florescu, D. Koller, and A. Levy. Using probabilistic information in data integration. In *Proceeding of the International Conference on Very Large Data Bases* (VLDB), 1997.

[FW97] M. Friedman and D. Weld. Efficiently executing information-gathering plans. In *Proceeding of the International Joint Conference of Artificial Intelligence (IJCAI)*, pages 785-791, 1997.

[FLMS99] D. Florescu, A. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *Proc. SIGMOD*, 1999.

[FMRU99] M. Franklin, G. Mihaila, L. Raschid, T.Urhan, M.E. Vidal, V. Zadorozhny. Search and Query Wide-Area Distributed Collections. Presented at the 1*999 Russian National Conference on Digital Libraries*, St. Petersburg, October 1999.

[GRZ$^+$00] Jean-Robert Gruser, Louiqa Raschid, Vladimir Zadorozhny, Tao Zhan: Learning Response Time for WebSources Using Query Feedback and Application in Query Optimization. VLDB Journal 9(1): 18-37 (2000)

[HKWY97] L. Haas, D. Kossman, E.L. Wimmers, J. Yang. Optimizing queries across diverse data sources. In *VLDB Conference*, 1997.

[IFF$^+$ 99] Zachary G. Ives, Daniela Florescu, Marc A. Friedman, Alon Y. Levy, and Daniel S. Weld. An adaptive query execution system for data integration. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, pages 299-310. ACM Press, 1999.

[KG99] S. Kambhampati and S. Gnanaprakasam. Optimizing source-call ordering in information gathering plans.*Proc. IJCAI-99 Workshop on Intelligent Information Integration*, 1999.

[KMA$^+$ 98] C. Knoblock, S. Minton, J. Ambite, N. Ashish, P. Modi, I. Mushlea, A. Philpot, and S. Tejada. Modeling web sources for information integration. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*, 1998.

[KW96] C. Kwok, and D. Weld. Planning to gather information. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, 1996.

[LK98] E. Lambrecht and S. Kambhampati. Optimizing information gathering plans. *Proc. AAAI-98 Workshop on Intelligent Information Integration*, 1998.

[LKG99] E. Lambrecht, S. Kambhampati and S. Gnanaprakasam. Optimizing recursive information gathering plans. In *Proceeding of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1999.

[LRO96] A. Levy, A. Rajaraman, J. Ordille. Query Heterogeneous Information Sources Using Source Descriptions. In *VLDB Conference*, 1996.

[MRV00] George A. Mihaila, Louiqa Raschid, Maria-Esther Vidal: Using Quality of Data Metadata for Source Selection and Ranking. WebDB (Informal Proceedings) 2000: 93-98

[Nau98] F. Naumann. Data fusion and data quality. In *Proc. of the New Techniques & Technologies for Statistics Seminar* (NTTS). Sorrento, Italy. 1998.

[NLF99] F. Naumann, U. Leser, J. Freytag. Quality-driven Integration of Heterogeneous Information Systems. In *VLDB Conference* 1999.

[OV99] M.T. Ozsu and P. Valduriez. Principles of Distributed Database Systems (2nd Ed). Prentice Hall. 1999.

[PL00] Rachel Pottinger , Alon Y. Levy , A Scalable Algorithm for Answering Queries Using Views Proc. of the Int. Conf. on Very Large Data Bases(VLDB) 2000. [SACL79] P. Selinger, M.

Astrahan, D. Chamberlin, R. Lorie, T. Price. Access path selection in a relational database management system. In *SIGMOD* 1979.

[YLUG99] R. Yerneni, C. Li, J. Ullman and H. Garcia-Molina. Optimizing large join queries in mediation systems. In *Proc. International Conference on Database Theory*, 1999.

[VP98] V. Vasslos, Y. Papakonstantinou. Using Knowledge of Redundancy for Query Optimization in Mediators.*Proc. AAAI-98 Workshop on Intelligent Information Integration*, 1998.

[WS96]Wang, R.Y. and D.M. Strong. Beyond accuracy: What data quality means to data consumers. Journal on Management of Information Systems. 12(4).

[YPGM98] Ramana Yerneni, Yannis Papakonstantinou, and Hector Garcia-Molina. Fusion queries over internet databases. In *Proceeding of the 6th Int. Conf. on Extending Database Technology (EDBT)*, Valencia, Spain, 1998.