Joint Optimization of Cost and Coverage of Query Plans in Data Integration

Zaiqing Nie & Subbarao Kambhampati
Department of Computer Science and Engineering
Arizona State University,Tempe, AZ 85287-5406
{nie,rao}@asu.edu

ABSTRACT

Existing approaches for optimizing queries in data integration use decoupled strategies–attempting to optimize coverage and cost in two separate phases. Since sources tend to have a variety of access limitations, such 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. 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 a simulation study that demonstrates that the plans generated by our approach will be significantly better, both in terms of planning cost, and in terms of plan execution cost, compared 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 [LRO96, DGL00, LKG99, 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 stored in the same primary database, and that the relation can always be accessed "in-whole." In contrast, in data integration scenarios, the optimizer sits at the mediator, and the relations are effectively stored across multiple and potentially overlapping sources, each of which may only contain a partial extension of the relation. The sources have a variety of access limitations–in terms of binding pattern restrictions on feasible calls and in terms of

the number of disjunctive selections that can be passed in a single query. Finally, and more importantly, 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 data 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. Existing approaches for optimizing queries in data integration [LRO96; NLF99; DL99; 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, they select top N plans from the ranked list and execute them. 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* inefficient plans.

In this paper we present techniques for joint optimization of cost and coverage of the query plans in data integration. Our algorithms search in the space of "parallel" query plans that support parallel access to multiple sources for each subgoal conjunct. An important advantage of parallel plans over linear plans is that they avoid the significant redundant computation inherent in executing all feasible linear plans separately. The plan generation process takes into account the potential parallelism between source calls, and the binding compatibilities between the sources included in the plan.

The rest of the paper is organized as follows. Section 2 uses a simple example to provide a brief survey of existing work on query optimization in data integration, as well as to motivate the need for our joint optimization approach. Section 3 discusses the syntax and semantics of the parallel query plans. Section 4 discusses the models for estimating the cost and coverage of parallel plans, and also discusses the specific methodology we use to combine cost and coverage into an aggregate utility metric. Section 5 describes two algorithms to generate parallel query plans and analyzes their complexity. Section 6 presents a comprehensive empirical evaluation which demonstrates that our approach can offer high utility plans (in terms of cost and coverage) for a fraction of the planning cost incurred by the existing approaches that use phased optimization with linear plans. This work is part of HAVASU, an ongoing project to develop a flexible query processing framework for data integration.

2. BACKGROUND AND MOTIVATION

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: S_{11}, S_{12}, S_{13}, each of which contain tuples for the **book** relation, two sources S_{21}, S_{22} each of

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

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

which contain tuples for the **price-of** relation, and two sources S_{31}, 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. 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.

In the following, we briefly discuss the limitations of existing approaches in optimizing this query, and motivate our approach.

Bucket Algorithm [LRO96]: The bucket algorithm by Levy et al. [LRO96] will generate three buckets, each containing the sources relevant to one of the three subgoals in the query:

> Bucket B(for **book**): S_{11}, S_{12}, S_{13}
> Bucket P(for **price-of**): S_{21}, S_{22}
> Bucket R(for **review-of**): S_{31}, 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 (using containment checks), 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:

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

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 "feasible" 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 [FKL97; NLF99; DL99] 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"), and then executing the top N plans , , 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 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 $40, then in plan p_6 we have to query S_{13}, the costliest among the accessible sources, a thousand times because of its binding pattern restriction. The total cost of this plan will thus be more than 6×10^5. In contrast, a lower ranked plan such as p_4 ($= (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 costs for the same logical plan. In other words, we have to jointly optimize source-call ordering and source selection to get good query plans.

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 accessed in parallel (see Figure 1).

In our continuing example, looking at the six 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 subgoal order:*price-of →book → review-of*. So we can use the following two parallel plans to give all of the tuples that the six plans give in this example:

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

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
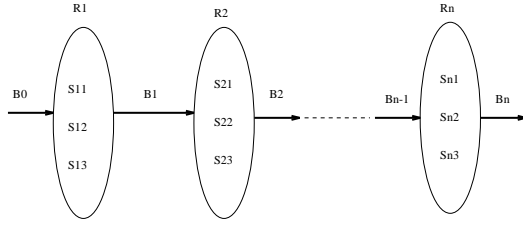
Figure 1: *A parallel query plan*

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 separate 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 parallel plans may be much larger than the space of linear plans–turns out to be a red herring. Specifically, our approach involves searching in the space of subgoal 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.

3. PRELIMINARIES

3.1 Schemas and queries

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. 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 query plans

A parallel query plan p has the form

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

Here sp_i is a subplan and S_{ij} is a source relation corresponding to the ith subgoal of the subgoal order used in the query plan. The 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 joins the results of the successive subplans to answer 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 a query 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 has as its instance 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.

In order to better illustrate the novel aspects of our joint optimization approach, we purposely assume a sequential execution model between subplans. However, it is entirely possible to provide a fully parallel (including pipelined) execution model for the plan that can reduce the time needed to generate first tuples [NK01].

4. COST AND COVERAGE MODELS

The main aim of this section is to describe the models we use to estimate the execution cost and coverage of (parallel) query plans, and how we combine the cost and coverage components into an aggregate utility metric for the plan.

Source Statistics: 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:

1. For each attribute A_i, its length, and for each attribute A_i in source relation S, the number of distinct values of A_i;

2. For each source: the number of tuples, the feasible binding patterns, the local delay time to process a query, the bandwidth between the source and the integration system and the initial set up latency;

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% of the tuples of the mediated 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.

These assumptions are in line with the types of statistics used by previous work (c.f. [LRO96,NLF99]). Techniques for learning response time statistics through probing are discussed in [GRZ$^+$00], while those for learning coverage statistics are developed in our recent work [NKNV01].

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 (although this assumption can be relaxed without affecting the advantages of our approach). Thus the execution costs

---

[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 executed in parallel (which reduces the execution time but incurs 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).

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

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 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 as the sum of the maximum response time of all the sources in the subplan and a fraction of the total response time of the sources in the subplan:
$$responseTime(sp_i) =$$
$$max_{j \in [1,m]}\{responseTime(S_{i_j}, B_{i-1})\}+$$
$$\beta \times \sum_{j \neq max_{RT}} responseTime(S_{i_j}, B_{i-1})$$
where β is a weight factor between 0 and 1, which depends on the level of parallelism assumed to be supported by the system and the network. $\beta = 0$ means that the system allows full parallel execution of all the sources queries, while $\beta = 1$ means that all the source queries have to be executed strictly sequentially. Notice that the response time of each source is being computed in the context of the binding relation preceding that source call. For a source S under the binding relation B, we have
$$responseTime(S, B) = msgDelay(S) * msgs(S, B)+$$
$$bytes(S, B)/localDelay(S)+$$
$$bytes(S, B)/bandWidth(S)$$
where $msgs(S, B)$ is the number of separate calls made to the source S under the binding relation B, and $bytes(S, B)$ denotes the total bytes sent back by the source S in response to these calls (with the remaining factors being part of the available source statistics). As explained in the extended paper [NK01], we can compute both $msgs(S, B)$ and $bytes(S, B)$ in terms of the available source statistics. In doing this, we assume that the mediator uses a variation of semi-join based strategy [OV99] to evaluate the joins of the source relations (the variation involves handling the source access restrictions).

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 p and the join selectivity factors of the subgoals and sources associated with these subplans. Let R_{sp_i} be the corresponding subgoal of the subplan $sp_i = \{S_{i_1}, S_{i_2}, ..., S_{i_m}\}$. 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 pairs of subgoal relations of the subplans in the plan, and $SF_J(sp_{i_1}, sp_{i_2})$ to denote the join selectivity factors of sources within the two subplans. Coverage of the plan p can be computed as:
$$coverage(p) = \frac{\prod_{i=1}^{n}[card(sp_i)] \times \prod_{i_1 < i_2}[SF_J(sp_{i_1}, sp_{i_2})]}{\prod_{i=1}^{n}[card(R_{sp_i})] \times \prod_{i_1 < i_2}[SF_J(R_{sp_{i_1}}, R_{sp_{i_2}})]}$$

If we assume that the subplans cover their respective relations uniformly (as is likely to be the case as the size of the subplans and their coverage increases), then $SF_J(sp_{i_1}, sp_{i_2}) = SF_J(R_{sp_{i_1}}, R_{sp_{i_2}})$. This, together with the fact that $\frac{card(sp_i)}{card(R_{sp_i})}$ is just the definition of $P(sp_i|R_{sp_i})$, simplifies the expression for coverage of p to
$$coverage(p) = \prod_{i=1}^{n}[P(sp_i|R_{sp_i})]$$

The coverage of a subplan itself can be written in terms of the coverages provided by the individual sources exporting that relation:
$$P(sp_i|R_{sp_i}) = P(\cup_{S_{i_j} \in sp_i} S_{i_j}|R_{sp_i})$$
$$= P(S_{i_1}|R_{sp_i}) + P(S_{i_2} \wedge \neg S_{i_1}|R_{sp_i}) + ...$$

$$+ P(S_{i_m} \wedge \neg S_{i_1} \wedge ... \wedge \neg S_{i_{m-1}}|R_{sp_i})$$
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

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]
$$utility(p) = w * log(coverage(p)) - (1 - w) * cost(p)$$
The logarithm ensures that the coverage contribution of a set of subgoals to the utility factor will be additive. The user can vary w from 0 to 1 to change the relative weight given to cost and coverage.[4]

5. GENERATING QUERY PLANS

The algorithms presented in this section aim to find the best parallel plan–i.e., the parallel plan with the highest utility[5]. 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 5.1. We then tackle the question of searching the space of subgoal orders. For this, we develop a dynamic programming algorithm (Section 5.2) as well as a greedy algorithm (Section 5.3).

5.1 Subplan Generation

The algorithm *CreateSubplan* shown in Algorithm 1 computes the best subplan for a subgoal R, given the statistics about the m sources $S_1, S_2, ..., S_m$ that export R, and the binding relation at the end of the current (partial) plan,

CreateSubplan 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 becomes less than the utility of the previous subplan. We use the models discussed in Section 4 to calculate the utility (cost and coverage) of the subplans.

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

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

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

[5] While such a 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 generate and execute a union of parallel plans. However, as we discuss in [NK01], this will not be a major issue in practice.

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 util-
       ity(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
```

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₌₁ᵐ 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:         if (w = 1) and (p′.coverage = p₁.coverage) and
             (p′.cost ≤ p₁.cost) then
24:           remove p₁ from S and push p′ into S
25:         else
26:           ignore p′
27:         end if
28:       else
29:         push p′ into S;
30:       end if
31:     end for
32:     p ← pop the first plan from S;
33:   end while
34:   return p;
35: end
```

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.

5.2 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. While our algorithm is related to a traditional system-R style algorithm, as well as its recent extension to handle binding pattern restrictions (but without multiple overlapping sources), given in [FLMS99], there are some important differences:

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

2. *ParPlan-DP* has 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 of the attributes in the binding relation.

3. *ParPlan-DP* needs to remember all the best partial plans for every subset of one or more of the n subgoals. For each subset, it stores the following information: (i) the best plan for these subgoals; (ii) the binding relation of the best plan; and (iii) the cost, coverage and utility of the 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 single subgoals are computed, followed by the best plans for pairs and larger subsets of subgoals, until the best plan for n subgoals is computed. 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. In practice, the algorithm does not need to generate all possible permutations. Permutations involving subgoals without any feasible source queries are eliminated.

Complexity: The worst case complexity of query planning with *ParPlan-DP* 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*.

We also note that our way of searching in the space of parallel plans does not increase the complexity of our query planning algorithm significantly. 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).

5.3 A Greedy Approach

We noted that *ParPlan-DP* already has better complexity 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 whose subplan can increase the utility of the current plan maximally. The subplan for that chosen subgoal is added to the plan, and the procedure is repeated until every subgoal has been covered.

Algorithm 3 ParPlan-Greedy

1: **Input**: $BUCKETS$: Buckets with n subgoals;
2: **output**: p : the best plan
3: **begin**
4: $B \leftarrow B_0$;
5: $UCS \leftarrow$ *all n subgoals in the query*;
6: $p \leftarrow \{\}$;
7: **while** $(UCS \neq \{\})$ **do**
8:    **for** each feasible subgoal $R_i (\in UCS)$ **do**
9:       $sp_i \leftarrow CreateSubplan(B, R_i)$;
10:    **end for**
11:    $sp_{max} \leftarrow$ subplan which will maximize $utility(p + sp_i)$ {If $w = 1$, among the subplans with the highest coverage, we choose the subplan with cheapest cost.};
12:    $p \leftarrow p + sp_{max}$;
13:    $UCS \leftarrow UCS - \{R_{max}\}$;
14:    $m \leftarrow$ # of sources in sp_{max};
15:    $B \leftarrow B \bowtie (\bigcup_{i=1}^{m} S_i)$; $\{S_i \in sp_{max}\}$
16: **end while**
17: **return** p;
18: **end**

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 6 that its performance may be quite fair in practice. It is of course possible to use this *ParPlan-Greedy* in concert with a hill-climbing([NK01]) approach to iteratively improve the utility of the query plan.

6. EMPIRICAL EVALUATION

We have implemented the query planning algorithms described in this paper. In this section, we describe the results of a set of simulation studies that we conducted with these algorithms. The goals of the study are: (i) to compare the planning time and estimated quality of the solutions provided by our algorithms with the approaches that enumerate and execute all linear plans, (ii) to demonstrate that our algorithms are capable of handling a spectrum of desired cost-quality tradeoffs, and (iii) to compare the performance of *ParPlan-DP* and *ParPlan-Greedy*. Towards the first goal, we implemented the approach described in [LRO96]. This approach enumerates all linear plans, finds feasible execution orders for all of them, and executes them.

The experiments were done with a set of simulated sources. The sources are specified solely in terms of the source statistics. We used 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 corresponding 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 to represent both slow and fast sources: 20% of sources have a high response time, 20% of them have low response time, and 60% of them have a medium response time. The source statistics are used to estimate the costs and coverages of the plans, as described in Section 3. The queries used in our experiments are
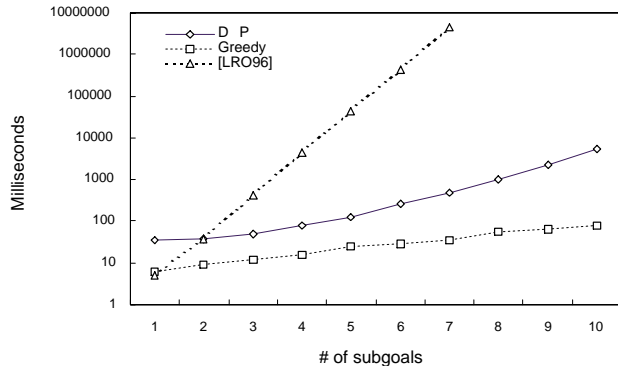

Figure 2: *Variation of planning time with the query size (when the the number of relevant sources per subgoal is held constant at 8). X axis plots the query size while Y axis plots the planning time.*

hybrid queries with both chain query and star query features[PL00], and their subgoals have 2-3 attributes. For example,

$$Q(A_1, A_4, A_7) : -R_1(A_1, A_2, A_3), R_2(A_2, A_3, A_4),$$
$$R_3(A_3, A_5, A_6), R_4(A_6, A_7), A_4 = x0.$$

The comparison between our approach and that in [LRO96] will be influenced by the parameter β. When β is close to 1, the amount of parallelism supported is low. This is particularly hard on the approach in [LRO96] as all the linear plans have to be essentially sequentially executed. Because of this, in all our simulations (except those reported in Figure 5), we use $\beta = 0$ as the parameter to compute the response times as this provides the maximum benefit to the algorithms in [LRO96], and thus establishes a lower bound on the improvements offered by our approach in comparison. All our simulations were run under JDK 1.2.2 on a SUN ULTRA 5 with 256Mb of RAM.

Planning time comparison: Figure 2 and Figure 3 compare the planning time for our algorithms with the approach in [LRO96]. In Figure 2, we keep the number of sources per subgoal constant at 8, and vary the number of subgoals per query from 1 to 10. In Figure 3, we keep the number of subgoals constant at 3, and vary the number of sources per subgoal from 5 to 50. 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 their best parallel plan. We see right away that both our algorithms incur significantly lower plan generation costs than the decoupled approach used in [LRO96]. We also note that *ParPlan-Greedy* scales much better than *ParPlan-DP* as expected.

Quality comparison: In Figure 4, we plot the estimates of the cost and coverage of plans generated by *ParPlan-DP* as a percentage of the corresponding values of the plans given by the algorithm in [LRO96]. The cost and coverage values for the plans generated by each of the approaches are estimated from the source statistics, using the methods described in Section 3. We use queries with 4 subgoals and each subgoal with 8 sources. Notice that the algorithms 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 approach remains the same for all values of w. We note that the best plan returned by our algorithm has a pretty high estimated 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
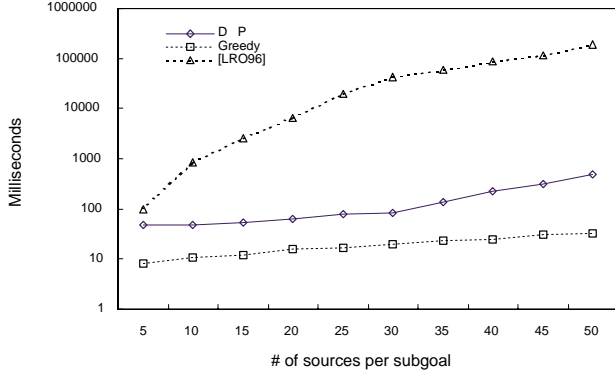
Figure 3: *Variation of planning time with number of relevant sources per subgoal (for queries of size 3). X axis plots the query size while Y axis plots the planning time.*
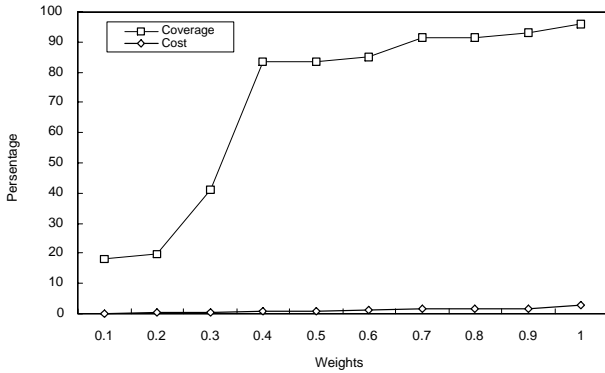

Figure 4: *Comparing the quality of the plans generated by* ParPlan-DP *algorithm with those generated by [LRO96] (for queries of 4 subgoals), while the weight in the utility measure is varied. X axis shows the weight value in the utility measure and Y axis plots the cost and coverage of our algorithm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

preference and attempting full coverage).[6] In Figure 5, we compare the execution cost of plans given by our approach with that given by [LRO96] with different values of β in the response time estimation. As we can see the bigger the β, the better our plan execution cost relative performance. This is because, for any β larger than 0, the cost model will take into account the cost of redundant queries, which will further worsen the execution cost of the [LRO96].

Comparing the greedy and exhaustive approaches: In order to compare *ParPlan-DP* and *ParPlan-Greedy* in terms of the plan quality, 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 6 shows the utility of the best plan produced by *ParPlan-Greedy* as a percentage of the utility of the plan produced by *ParPlan-DP*. We observe, as

---

[6] It is interesting to note that the execution cost for our approach turns out to be better than that of [LRO96] even when $w = 1$, when both approaches are forced to access all sources to maximize coverage. Since we kept $\beta = 0$, one might think that the execution costs should be same for this particular case. The reason our approach winds up having better execution cost even in this scenario is that it finds the best execution order of the parallel plan. In contrast, the [LRO96] approach just finds a feasible execution order for its plans. Because there are so many linear plans, the probability that one of them will wind up getting a feasible plan with high execution cost is quite high.
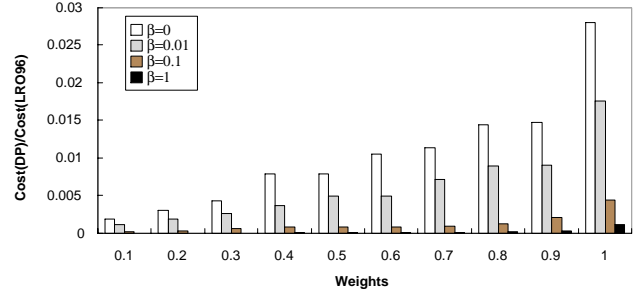

Figure 5: *Ratio of the execution cost of the plans given by* ParPlan-DP *to that given by [LRO96], for a spectrum of weights in the utility metric and parallelism facter β in the response time estimate. X axis varies the coverage-cost tradeoff weights used in the utility metric, and Y axis shows the ratio of execution costs for different β.*
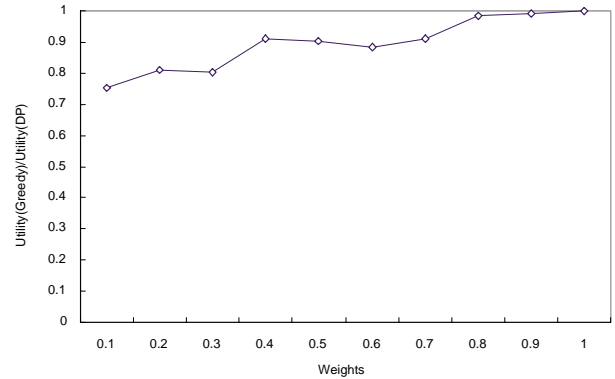

Figure 6: *Ratio of the utility of the plans given by* ParPlan-Greedy *to that given by* ParPlan-DP *for a spectrum of weights in the utility metric. X axis varies the weight used in the utility metric, and Y axis shows the ratio of utilities.*

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*.

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 7 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. 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.

7. RELATED WORK

Bucket algorithm [LRO96] and the source inversion algorithm [DGL00] provide two of the early approaches for generating candidate query plans in data integration. As we mentioned in Section 2 the disadvantages of generating all possible linear plans and executing them have lead to other alternate approaches. The work on
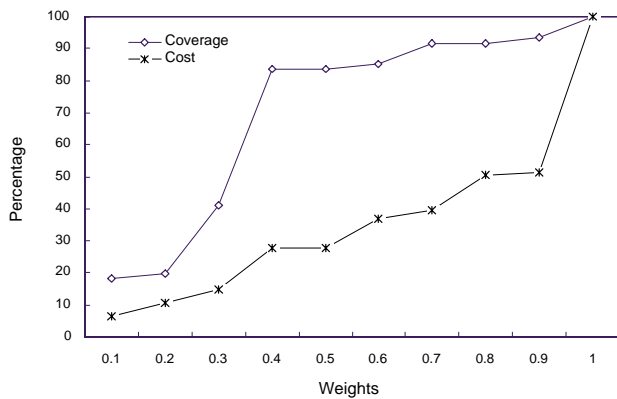
Figure 7: *Comparing the Coverage and cost of the plans found by ParPlan-DP by using different weights in Utility function, on queries of 4 subgoals. X axis varies the weights in the utility function, while the Y axis shows the cost and coverage as a percentage of the cost and coverage offered by our ParPlan-DP with weight=1.*

Streamer project [DL99] extends the query planning algorithm in [LRO96], so it uses the coverage information to decide the order in which the potential plans are executed. The coverage is computed using the source overlap models in [FKL97]. 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. Although their notion of quality seems to include both cost and coverage, their cost model seems to be quite restrictive, making their approach a phased one in essence. For example, they claim that "a change of the join execution order within a plan has no effect on its IQ [quality] score." As we have seen, join orders do have a significant impact on the overall quality (utility) of the plan.

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. Finally, parallel plans (or joins over unions) are quite standard in distributed/parallel database systems [LPR98; OV99]. Use of parallel plans in data-integration scenarios does however pose several special challenges because of the uncontrolled overlap between data sources, the source access (binding) restrictions, and the need to produce plans for a variety of cost/coverage requirements.

8. CONCLUSION

In this paper we started by motivating the need for joint optimization of cost and coverage of query plans in data 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. We also demonstrated the flexibility of our algorithms in handling a spectrum of cost-coverage tradeoffs.

The optimization algorithms presented in this paper are being integrated into a prototype system called HAVASU that we are developing for supporting query processing in data integration. *Havasu* system is intended to support multi-objective query optimization, flexible execution strategies for parallel plans, as well as mining strategies for learning source statistics [NKNV01].

References

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

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

[DGL00] Oliver M. Duschka, Michael R. Genesereth, Alon Y. Levy. Recursive Query Plans for Data Integration. In *Journal of Logic Programming, Volume 43(1)*, pages 49-73, 2000.

[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.

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

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

[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.

[LPR98] Ling Liu, Calton Pu, Kirill Richine. Distributed Query Scheduling Service: An architecture and its Implementation. In *Special issue on Compound Information Services, International Journal of Cooperative Information Systems (IJCIS)*. Vol.7, No.2&3, 1998. pp123-166. 1998.

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

[NK01] Z. Nie, S. Kambhampati. Joint Optimization of Cost and Coverage of Information Gathering Plans. ASU CSE TR 01-002. Computer Science & Engg. Arizona State University.
`http://rakaposhi.eas.asu.edu/havasu.html`.

[NKNV01] Z. Nie, S. Kambhampati, U. Nambiar and S. Vaddi. Mining Source Coverage Statistics for Data Integration. Proc. WIDM 2001.

[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.