Efficiently Ordering Query Plans for Data Integration

AnHai Doan and **Alon Levy**
Box 352350
Department of Computer Science and Engineering
University of Washington, Seattle, WA, 98195
{anhai,alon}@cs.washington.edu

Abstract

We describe Streamer, the query-reformulation component of a data integration system. Given a utility measure and a user query, Streamer uses abstraction-based refinement planning and exploits information on plan independence to produce, in decreasing order of utility, a set of plans that access data sources to obtain answers to the query. We then focus on plan coverage as an important utility measure. We show how to use statistic information about the domain and data sources to estimate plan coverage, and how to incorporate the plan-coverage framework into Streamer. In doing so, we provide the first method for effectively integrating the use of quantitative information into the query optimizer of a data-integration system. We present preliminary experimental results suggesting that Streamer runs an order of magnitude faster than brute-force plan-ordering methods, which are the only currently available methods to compute *exact* plan orderings. Finally, we propose methods to make Streamer scalable to large domains.

Introduction

The problem of data integration has recently received significant attention in both the AI and database research communities, as evidenced by numerous projects on developing data-integration systems (Chawathe *et al.* 1994; Knoblock *et al.* 1998; Levy, Rajaraman, & Ordille 1996; Friedman & Weld 1997). The goal of a data-integration system is to answer user queries using a variety of data sources: given a user query, the system creates a query-execution plan that identifies sources relevant to the query, and determines how the data from the different sources are to be combined.

As a typical example, consider the domain of movies. Suppose we have three sources of type A, each containing a listing of movie titles and actor names, and two sources of type B, each containing a listing of movie titles and their reviews. Given a query asking for reviews of movies starring Harrison Ford, a data-integration system may generate and execute six plans, each accessing a source of type A to ask for the titles of movies starring Ford, then feeding these titles into a source of type B to obtain the reviews. The *combined output* of these six plans is then returned as the answer to the query. We emphasize that unlike AI planning, where we need to execute only a *single* plan from a set of alternative plans, in data integration the answer to a query is the *union* of the output of all plans that answer the query, and hence we need to execute more than one plan (although in some cases we actually don't need all of them to give the users what they really want).

Given a number of plans that answer a query, we would like to *order* them so that we can focus on and execute the most relevant plans first. A motivation for this is the case of *plan coverage*, defined as the number of new tuples returned by the plan that satisfy the user query. Executing plans in decreasing order of their coverage tends to maximize the likelihood of obtaining answers early (Florescu, Koller, & Levy 1997), as well as reduce the amount of retrieved redundant information – thus saving computational resources. Another example is execution time. Executing plans in the increasing order of their execution time tends to minimize response time.

The problem of ordering plans poses difficult computational challenges. Many data-integration domains – especially those on the Web – comprise a large number of sources, resulting in a huge number of plans. This requires that a plan-ordering algorithm deal effectively with a large number of sources to guarantee acceptable performance. To address this problem, we introduce the use of a planning technique known as abstraction-based refinement planning (ARP) to the problem of generating and ordering query execution plans. ARP as embodied in the Drips planner (Haddawy, Doan, & Goodwin 1995) is concerned with finding the plan with the highest utility from a given set of plans. Drips groups together actions (sources in our case) with similar characteristics so it can create and compare abstract plans during the planning process. Each abstract plan represents a set of concrete plans; the power of Drips comes from the fact that it is able to prune significant chunks of the plan space by eliminating some abstract plans from consideration. As such, ARP is well suited to the problem of selecting a plan among competing ones. However, in data integration a single plan does not give all the answer; we also need to be able to find the next best plans because the real answer is the union of all plans. ARP thus is not suited for data integration.

To further adapt ARP to the data-integration context, we make the following contributions:

- First, we describe Streamer, a plan-ordering algorithm that extends Drips such that it can also efficiently produce the *next* best plan, as opposed to only a single best plan. Streamer keeps track of dominance relations among abstract plans that are created while searching for the best plan. The dominance relations and plan-independence information are then used to considerably prune the search space for the next best plans.
- Second, we show how to combine ARP with quantitative information about the data sources and the domain to estimate the expected coverage of a query plan. In doing so we provide the first method for effectively integrating the use of quantitative information (originally suggested in (Florescu, Koller, & Levy 1997)) to the query optimizer of a data-integration system.
- Third, we present preliminary experimental results suggesting that Streamer runs an order of magnitude faster than brute-force plan-ordering methods, which – as far as we know – are the currently best methods to compute *exact* plan orderings.
- Finally, we propose methods to make Streamer scalable to large domains.

Background & Problem Definition

In this section we introduce a data-integration model together with the concept of plan coverage, then formally define the plan-ordering problem.

Mediated-Schema and Source Relations: We model a data-integration domain with a number of *mediated-schema relations* (or *schema relations* for short). For example, the movie domain described in the introduction can be modeled with the schema relations in Figure 1. Schema relations are *virtual* in that they are only meant to provide the user with a uniform interface to a multitude of data sources, and that we do not actually store the extensions of these relations.

We model the contents of the external data sources with a set of *source relations* disjoint from the schema relations. A source relation is described by a *source description* in terms of a conjunction of schema relations. The meaning of such a source description is that all the tuples that are found in the source satisfy the schema relations in the conjunction. For example, the source description $V_1(A, M)$:– play-in(A, M), american(M) in Figure 1 says that this source stores a relation V_1 with tuples of the form $\langle A, M\rangle$ such that actor A plays in the American movie M. Figure 1 lists the source descriptions of four movie-domain data sources.

User Queries and Conjunctive Plans: A *user query* has the form $Q(\overline{Y})$:– $R_1(\overline{Y_1}), \ldots, R_m(\overline{Y_m})$, where R_i are schema relations, and $\overline{Y_i}$ and $\overline{Y}$ denote tuples of variables. We require that $\overline{Y} = \cup_{i=1}^{m} \overline{Y_i}$, which implies that the data-integration model has no projection capabilities. Semantically, query $Q(\overline{Y})$ asks for all data tuples of type $\overline{Y}$ that can be obtained from data sources and satisfy the relations $R_1, \ldots, R_m$. For example, a query asking for all tuples $\langle A, M, R\rangle$ such that R is a review of movie M starring actor A might be formulated as the sample query shown in Figure 1.

Schema relations:	**Source relations:**
play-in(A,M)	V_1(A,M) :- play-in(A,M), american(M)
review-of(R,M)	V_2(A,M) :- play-in(A,M), russian(M)
american(M)	V_3(R,M) :- review-of(R,M)
russian(M)	V_4(R,M) :- review-of(R,M)

Sample query:

Q(A,M,R) :- play-in(A,M), review-of(R,M)

Bucket B1	**Bucket B2**
{V_1, V_2}	{V_3, V_4}

Figure 1: The data-integration domain of movies

A *conjunctive plan* p (or simply *plan* p when there is no ambiguity) that answers a query Q returns tuples that satisfies Q and has the form p :– $V_1(\overline{X_1}) \ldots V_m(\overline{X_m})$, where each V_i is a source relation corresponding to a data source. The meaning of plan p is that it accesses and produces the join of tuples from the sources $V_1(\overline{X_1}), \ldots, V_m(\overline{X_m})$ as the answer. For example, p :– $V_1 V_3$ is a conjunctive plan that answers the query in Figure 1. The answer to a user query is the union of the tuples returned by all conjunctive plans[1]. Notice that in our data-integration model for the ease of exposition we have assumed no projection and selection capabilities.

Generating Conjunctive Plans: Given a query we can generate conjunctive plans to answer it using the *bucket algorithm* described in (Levy, Rajaraman, & Ordille 1996). For each schema relation R in the query this algorithm creates a *bucket*, which is the set of all sources that can return tuples that satisfy R. It then combines sources from the buckets to form conjunctive plans. For example, for query Q in Figure 1 the bucket algorithm would create two buckets B_1 and B_2 as shown in that figure. It would then combine sources, one from each bucket, to form four plans: V_1V_3, V_1V_4, V_2V_3, and V_2V_4.

Creating the buckets can be done efficiently. So in the rest of this paper, given a query we assume the buckets have been created, and we are concerned only with ordering plans that are created by taking the Cartesian product of the buckets.

[1] The use of the term *plan* in this context is a bit troublesome. Plans have different meanings in the AI and database literatures, *neither* of which matches the meaning in the data integration literature. In the AI literature, plans also affect the environment, which does not happen here. In the database literature, plans are imperative programs that tell the query execution engine exactly how to evaluate a query. In our case, a plan is a way of obtaining the answers to a query from the data sources, but, the plans still have to be optimized (and hence, turned into plans in the database sense) before they can be executed. In the optimization phase, the order of the conjucts will be determined, as will the physical algorithms used to implement joins, unions, and other relational operators and the resource allocation to each one of the operators in the plan.

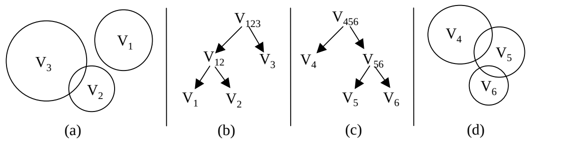

Figure 2: Sources and abstract actions in two buckets

Utility, Plan Coverage, and Plan Ordering: Informally, we define the *utility* of a plan p with respect to a set of plans $\{p_1, \ldots, p_n\}$ and a query Q to be a real number that indicates the relative "worth" of p given that plans $p_1, \ldots, p_n$ have been executed. We consider only utility measures that are *monotonic* in the sense that plan utility does *not* increase when new plans are introduced into the set of plans that already have been executed. This definition includes almost all of the utility measures that have been considered in AI planning and in data integration.

An example of such a utility measure is *plan coverage*. We define *the coverage of a plan p* with respect to a set of plans $\{p_1, \ldots, p_n\}$ and a query $Q(\overline{Y})$ to be the probability that a tuple chosen randomly among those tuples of type $\overline{Y}$ that satisfy Q will be returned by plan p and not by any plan p_i, $i \in [1, n]$. For example, assume that there are 10,000 tuples of type $\langle A, M, R\rangle$ that satisfy query Q in Figure 1. Assume further that we pick $p_1 :- V_1 V_3$ as the very first plan for execution, and that p_1 returns 1,000 tuples. Then the coverage of p_1 is $1,000/10,000 = 0.1$. Now assume that we pick $p_2 :- V_2 V_4$ as the next plan, and that p_2 returns 2,000 tuples, out of which 500 tuples have already been returned by p_1. Then the coverage of p_2 with respect to p_1 and Q is $(2,000 - 500)/10,000 = 0.15$.

We now define the plan-ordering problem as follows: **Definition (Plan-Ordering Problem):** *Assume for a query Q the bucket algorithm generates the buckets $B_1, \ldots, B_m$, the Cartesian product of which forms a set of k conjunctive plans. Find an ordering $\{p_1, \ldots, p_k\}$ of these plans such that for any $i \in [1, k]$ the utility of p_i with respect to $p_1, \ldots, p_{i-1}$ is the highest among the utilities of plans $p_i, \ldots, p_k$.*

This problem is a special case of a more general plan-ordering problem considered in (Florescu, Koller, & Levy 1997), which finds the best k plans to be executed in parallel at a given point, with k being a prespecified value that best utilizes the network bandwidth.

Abstraction-Based Planning

We can use the abstraction-based planner Drips to find the first plan in the plan ordering (i.e., the one with the highest utility). The basic idea of this algorithm is to group and abstract sources so that it can create abstract plans during the planning process. Each abstract plan represents a set of concrete (conjunctive) plans, and has as its utility a *real-valued interval* that contains the utility of *all* concrete plans in the set. We say that a plan p *dominates* a plan q if there is at least one concrete plan $p' \in p$ whose utility is not less than the utility of all concrete plans in q. During the planning process, if Drips finds a plan pair (p, q) with the respective utility intervals $[l_p, h_p]$ and $[l_q, h_q]$ such that $l_p \geq h_q$ then Drips eliminates q, and thus all concrete plans represented by q, from any further consideration. Notice that the elimination of q takes place without having to explicitly compute the utility of the concrete plans associated with it, hence the computational savings. Drips iteratively creates, refines, and eliminates abstract plans until only the best concrete plan remains.

We explain Drips with the following example. Suppose the utility is plan coverage and the set of plans is given as the Cartesian product of two buckets, each consisting of three sources as shown in Figures 2.a and 2.d. In these figures sources are represented with circles, the overlaps of which mean overlaps of the actual sources.

We iteratively group and abstract sources in each bucket. For the bucket in Figure 2.a, we abstract two sources V_1 and V_2 into the abstract source V_{12}, which is then abstracted with V_3 to yield the final abstract action V_{123} for this bucket[2], as shown in Figure 2.b. A similar abstraction process is shown in Figure 2.c for the second bucket. The abstract plan $V_{123}V_{456}$ represents all nine concrete plans $\{V_1, V_2, V_3\} \times \{V_4, V_5, V_6\}$.

We start planning by refining the top abstract plan $V_{123}V_{456}$ into a set of lower-level abstract plans by replacing an abstract source in this plan with its component sources. Lets say we pick V_{123} and refine it into the two component sources V_{12} and V_3, yielding two plans $V_{12}V_{456}$ and V_3V_{456}.

Assume we compute the coverage of these two plans to be [0.1,0.7] and [0.5,0.8], respectively. Since these intervals overlap, we cannot eliminate either plan. We then pick a plan, say $V_{12}V_{456}$, and refine it into V_1V_{456} and V_2V_{456}. Assume the coverage of these plans are [0.4,0.6] and [0.1,0.3], respectively, then we can eliminate V_2V_{456} because it is dominated by V_1V_{456}.

We now have two plans left: V_3V_{456} and V_1V_{456}. We pick V_3V_{456} and refine it into V_3V_4 and V_3V_{56}. Assume the coverage of these plans are 0.8 and [0.6,0.7], respectively. Then V_3V_4 dominates V_3V_{56} and V_3V_{56} in turn dominates V_1V_{456}, and we can eliminate both dominated plans. Since we now have only plan V_3V_4 left and it is a concrete plan, we return V_3V_4 as the plan with the highest coverage, and as the first plan in the ordering.

An Algorithm for Ordering Plans

Drips is not suited for data integration because it finds only the first plan in the ordering. In this section we describe Streamer which finds the entire plan ordering. Streamer uses abstraction techniques as in Drips

[2]Source abstraction in general can be done efficiently, using a variety of heuristics. Space limitations preclude a thorough discussion of this topic, but see (Haddawy, Doan, & Goodwin 1995) for more details, and see also the experiments section for a simple abstraction heuristic.

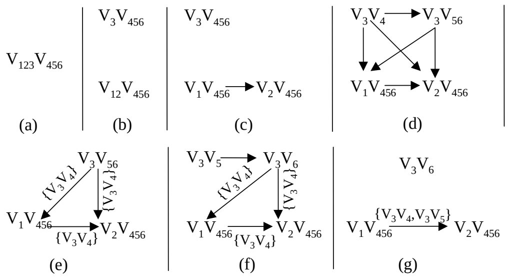

Figure 3: Stages of the domination graph created by Streamer on the example in Figure 2. The set associated with a link is shown next to that link.

to efficiently find the next plan in the ordering. But unlike Drips, Streamer also keeps track of dominance relations among the abstract plans created during the planning process. The dominance relations and plan-independence information are then used to considerably prune the search space for the next best plans.

The complete algorithm of Streamer is shown in Figure 4, and we shall explain its working via the example discussed in the previous section. Initially Streamer proceeds exactly like Drips: it starts with the top plan $V_{123}V_{456}$ and refines this plan into V_3V_{456} and $V_{12}V_{456}$ (Figures 3.a and 3.b, and Step 2.c of Figure 4). Next, it refines $V_{12}V_{456}$ into V_1V_{456} and V_2V_{456}. At this point unlike Drips which eliminates V_2V_{456} because it is dominated by V_1V_{456}, Streamer does not eliminate any plan; it simply creates a *domination link* from V_1V_{456} to V_2V_{456} (Figure 3.c and Step 2.b of Figure 4), denoted as $V_1V_{456} \rightarrow V_2V_{456}$. We associate each link $p \rightarrow q$ with a set (of plans) denoted as $E(p,q)$. When a link is initially created, its set of plans is empty and is not shown in Figure 3. We will describe shortly how this set is maintained and what purpose it serves. Streamer thus maintains a *dominance graph* whose nodes are plans and whose edges are domination links specifying dominance relationships.

Next, like Drips, Streamer picks V_3V_{456} and refines it into V_3V_4 and V_3V_{56} (Figure 3.d and Step 2.c of Figure 4). Recall from the previous section that V_3V_4 dominates V_3V_{56}, which in turn dominates V_1V_{456}. But again, instead of eliminating dominated plans as in Drips, Streamer creates dominance links pointing from dominating plans to dominated ones (Step 2.b). At this point, the only nondominated plan, V_3V_4, is a concrete plan, and so is returned as the best plan (line 1 of Step 2.d).

Once V_3V_4 has been returned, Drips terminates, but Streamer continues on to find subsequent plans. First, it removes V_3V_4 from the dominance graph, thus removing all links originating from this plan (Figure 3.e and line 1 of Step 2.d of Figure 4). However, removing a plan might change the utility of the remaining plans, thus rendering some domination links invalid. For example, after removing V_3V_4 the coverage of V_2V_4 will change because these two plans overlap[3]. So suppose initially V_2V_4 dominates a plan q and we have the link $V_2V_4 \rightarrow q$, then after removing V_3V_4, V_2V_4 may no longer dominate q and the link may become invalid.

After removing V_3V_4, Streamer needs to recheck the validity of each remaining link. To do this Streamer uses information on *plan independence*. We say that plan s is *independent* of plan t if the utility of s does not depend on whether t has been removed. For example, in the case of plan coverage, two plans are independent iff they do not overlap, which can be verified very easily by examining the overlap of their sources.

Streamer checks the validity of a link $p \rightarrow q$ by adding V_3V_4 to the set $E(p,q)$, then checking if there is a concrete plan $s \in p$ that is independent of all concrete plans in $E(p,q)$. If Streamer finds such a plan then it concludes that link $p \rightarrow q$ is still valid. This is true because (a) at the time link $p \rightarrow q$ was created, s dominates q; (b) since that time only plans in $E(p,q)$ have been removed, and s is independent of these plans, so the utility of s hasn't changed; and (c) the utility of any plan in q cannot increase[4].

Returning to our example, we can easily check that after removing V_3V_4 all three links shown in Figure 3.e are still valid. For example, link $V_3V_{56} \rightarrow V_1V_{456}$ is valid because plan $V_3V_6 \in V_3V_{56}$ is independent of plan V_3V_4 in $E(V_3V_{56}, V_1V_{456})$. This is so because V_6 and V_4 do not overlap.

Having checked the validity of links (line 2 of Step 2.d of Figure 4), Streamer picks a nondominated plan, say V_3V_{56} in Figure 3.e, and refines it into V_3V_5 and V_3V_6. Assume Streamer computes the coverage of these two plans to be 0.7 and 0.65. Then Streamer finds that V_3V_5 dominates V_3V_6 and so creates the link $V_3V_5 \rightarrow V_3V_6$ (Figure 3.f). Notice that the set E of this link is empty because the link was just created.

At this point, the only nondominated plan in the graph is the concrete plan V_3V_5, so Streamer returns it as the next best plan and removes it from the dominance graph. After that Streamer checks and removes invalid links as in the case of removing V_3V_4. The dominance graph after removing invalid links is shown in Figure 3.g. Link $V_1V_{456} \rightarrow V_2V_{456}$ is still valid because plan V_1V_6 is independent of both plans in the set E of that link. Other links are invalid and are removed from the graph. Streamer then continues on to find the next best plan, and so on.

Plan Coverage as Utility Measure

In this section, we show how to incorporate plan coverage into our improved ARP algorithm. Plan coverage was first discussed in (Florescu, Koller, & Levy 1997). In that work the authors describe how to use quantitative information about the data sources and the domain to estimate coverage of plans of length one. In this section we extend that framework to conjunctive

[3] This is due to the fact that V_3 and V_2 overlap (Figure 2)

[4] For example, after removing a plan r, the coverage of any plan t remains unchanged if t is independent of r, and decreases otherwise.

Input: Query Q, utility measure u, set of buckets $B_i, i \in [1, m]$; **Output:** A list of plans in $B_1 \times \ldots \times B_m$, in decreasing order of utility

1. **Foreach** B_i **do** create top abstract source T_i. Put top plan $P = T_1 \ldots T_m$ into a dominance graph G. Set $u(P) \leftarrow nil, dom(P) \leftarrow false$
2. Loop until G becomes empty:
   (a) **Foreach** nondominated p (i.e., $dom(p) = false$) in G st $u(p) = nil$ **do** recompute $u(p)$.
   (b) **Foreach** pair of nondominated $p, q \in G$, $u(p) = [l_p, h_p]$, $u(q) = [l_q, h_q]$, $l_p \geq h_q$ **do** create link $p \rightarrow q$; set $E(p,q) \leftarrow \emptyset$, $dom(q) \leftarrow false$.
   (c) **If** $\exists$ abstract, nondominated $p \in G$ **then** refine p into $p_1, \ldots, p_l$; set $u(p_i) \leftarrow nil, dom(p_i) \leftarrow false, G \leftarrow G \cup \{p_1, \ldots, p_l\}$.
   Foreach $p' \in G$ st $p \rightarrow p'$, $p_i, i \in [1, l]$, st $\mathsf{CheckValidity}(p_i, E(p, p')) = true$ **do** create $p_i \rightarrow p'$, set $E(p_i, p') \leftarrow E(p, p')$.
   Remove p from G. Go back to Step 2.a.
   (d) **If** not $\exists$ abstract nondominated $p \in G$ **then** choose any $b \in G$ and output b as the next best plan. Set $G \leftarrow G - \{b\}$.
   Foreach $q \rightarrow q'$ **if** $\mathsf{CheckValidity}(q, E(q, q') \cup \{b\}) = true$ **then** $E(q, q') \leftarrow E(q, q') \cup \{b\}$ **else** remove $q \rightarrow q'$.
   Foreach $p \in G$ st p and b are not independent **do** $u(p) \leftarrow nil$.

Algorithm $\mathsf{CheckValidity}$(plan c, set of concrete plans F): Return $true$ iff $\exists$ concrete $c' \in c$ st c' is independent of all concrete $f \in F$.

Figure 4: A high-level description of Streamer

plans. We use the same kinds of quantitative informations, with a little more. We then extend further our framework to handle abstract sources and plans so that it can work with Streamer.

Before going on, we introduce some notation. We write $N_{\overline{Y}}(E)$ to refer to the number of tuples of type $\overline{Y}$ that satisfy expression E. For example $N_{\overline{Y}}(R)$ and $N_{\overline{Y}}(p)$ refer to the number of tuples of type $\overline{Y}$ that satisfy relation $R(\overline{Y})$ and that are returned by plan p, respectively.

We write $P_{\overline{Y}}(E|F)$ to refer to the probability that a tuple chosen randomly among the tuples of type $\overline{Y}$ that satisfy expression F also satisfies expression E. In general, $P_{\overline{Y}}(E|F) = N_{\overline{Y}}(E)/N_{\overline{Y}}(F)$ only when F contains all tuples of type $\overline{Y}$ in E, which is the case for all such equations in this paper. Occasionally we shall drop the subscript $\overline{Y}$ if there is no ambiguity.

Finally, we write CQ_V to refer to the source description of a source with the source relation V. For example, for source $V_1(A, M)$:- $\mathsf{play\text{-}in}(A, M)$, $\mathsf{american}(M)$ in Figure 1 we write CQ_{V_1} to refer to the description $\mathsf{play\text{-}in}(A, M) \wedge \mathsf{american}(M)$.

Quantitative Information To estimate the coverage of conjunctive plans, we add quantitative information to the mediated schema and the data sources.

First, we specify the degree of *overlap between schema relations*. Specifically, for each schema relation $R(\overline{Y})$, we add information to estimate $P_{\overline{Y}}(Q|R)$, where Q is an arbitrary conjunction of schema relations.

Second, we specify the *coverage of a data source*. More specifically, for each source with relation $V(\overline{Y})$:- CQ_V, we add a number estimating $P_{\overline{Y}}(V|CQ_V)$ – the coverage of V wrt CQ_V.

The motivation for these two kinds of quantitative information is that they help estimate to what degree a source $V(\overline{Y})$ with description Q_V covers the set of tuples of type $\overline{Y}$ that satisfy a query conjunct $R(\overline{Y})$. This degree can be computed as

$$\begin{aligned} P_{\overline{Y}}(V|R) &= N_{\overline{Y}}(V)/N_{\overline{Y}}(R) \\ &= [N_{\overline{Y}}(V)/N_{\overline{Y}}(CQ_V)][N_{\overline{Y}}(CQ_V)/N_{\overline{Y}}(R)] \\ &= P_{\overline{Y}}(V|CQ_V)P_{\overline{Y}}(CQ_V|R). \qquad (1) \end{aligned}$$

Notice that the first term in (1) is simply the coverage of V wrt CQ_V, and the second term can be computed using the schema overlap information.

Third, we specify *overlaps between sources*. In the current framework, we can only express two types of overlap relationships: (a) two sources are disjoint; and (b) two sources overlap, but their contents are assumed to be independent of each other. Overlap information of type (a) can be used to establish plan independence that is exploited by Streamer.

Overlap information of type (b) can be used to compute the overlap between two sources. More specifically, given two overlapping sources $S(\overline{Y})$ and $V(\overline{Y})$, since we assume that their contents are independent of each other, $P_{\overline{Y}}(S, V|CQ_S, CQ_V)$ can be computed as follows

$$\begin{aligned} P(S, V|CQ_S, CQ_V) &= P(S|CQ_S, CQ_V)P(V|CQ_S, CQ_V) \\ &= P(S|CQ_S)P(V|CQ_V) \qquad (2) \end{aligned}$$

assuming that the content of any source is uniformly distributed within the description of that source.

Finally, we estimate the *number of distinct values of attributes* in schema and source relations. Specifically, for each schema or source relation $R(\overline{Y})$ and attribute $a \in \overline{Y}$, we add a term $d(a, R)$ that estimates the number of distinct values of a in R. In obtaining answers for a query, we need to join tuples from different sources, and this information helps in estimating the size of the join result.

The above quantitative framework is a generalization of that of (Florescu, Koller, & Levy 1997). In that framework, the authors considered only query plans of length one, where no joining is necessary. So they considered only the first three kinds of quantitative information described above. They also pointed out two challenges in dealing with quantitative information: how to obtain such information, and how to represent it compactly and to maintain information consistency. We note that the quantitative information discussed here can be obtained through a number of methods, including statistical sampling and analyzing sources that are considered complete. Space limitations preclude a thorough discussion, but see (Florescu, Koller, & Levy 1997; Vassalos & Papakonstantinou 1998) for proposed methods to handle these issues.

Estimating Plan Coverage: We now describe how to estimate the coverage of a conjunctive plan. Our estimation framework relies on two key observations:

- Given a plan p, its coverage $P_{\overline{Y}}(p|Q)$ can be expressed as the ratio $N_{\overline{Y}}(p)/N_{\overline{Y}}(Q)$; and
- Given $p :- V_1 \ldots V_m$, the set of tuples of type $\overline{Y}$ that are returned by p can be obtained by joining tuples from the sources $V_1, \ldots, V_m$. So we can estimate $N_{\overline{Y}}(p)$, the size of the join result, using whatever appropriate statistic information and estimation methods as employed in traditional-database query optimization (Selinger *et al.* 1979; Ramakrishnan 1998; Mannino, Chu, & Sager 1988; Lipton & Naughton 1990; Ioannidis 1993). The same is true for estimating $N_{\overline{Y}}(Q)$.

For example, consider the case when no plan has yet been chosen for execution. Then given a query $Q(\overline{Y}) :- R_1(\overline{Y_1}), R_2(\overline{Y_2})$ and a plan $p_1 :- V_1(\overline{Y_1})V_2(\overline{Y_2})$, we can compute

$$\begin{aligned} P_{\overline{Y}}(p_1|Q) &= N_{\overline{Y}}(V_1 \bowtie V_2)/N_{\overline{Y}}(R_1 \bowtie R_2) \\ &= [N(V_1)/N(R_1)][N(V_2)/N(R_2)]\mathcal{F}^1_V\mathcal{F}^1_R \\ &= P_{\overline{Y_1}}(V_1|R_1)P_{\overline{Y_2}}(V_2|R_2)\mathcal{F}^1_V\mathcal{F}^1_R \qquad (3) \end{aligned}$$

where $\mathcal{F}^1_V$ for example refers to the selectivity factor of joining V_1 with V_2, and can be computed as discussed in (Selinger *et al.* 1979), using information on the number of distinct values of attributes[5]. The first two terms in (3) can be computed as shown in (1).

We now consider how to compute the coverage of say $p_2 :- V_3(\overline{Y_1})V_4(\overline{Y_2})$ assuming that plan $p_1 :- V_1V_2$ has been chosen for execution. We write this coverage as

$$P(p_2, \overline{p_1}|Q) = P([V_3, \overline{V_1}]V_4|Q) + P([V_3, V_1][V_4, \overline{V_2}]|Q) \quad (4)$$

where $[V_3, \overline{V_1}]V_4$ forms a plan that joins all tuples that are in V_3 but not in V_1 with all tuples that are in V_4. We will show in a moment how to derive (4). For now, we consider how to compute $P([V_3, \overline{V_1}]V_4|Q)$. Applying (3) we can write

$$P([V_3, \overline{V_1}]V_4|Q) = P(V_3, \overline{V_1}|R_1)P(V_4|R_2)\mathcal{F}^2_V\mathcal{F}^2_R \quad (5)$$

where $\mathcal{F}^2_V$ for example refers to the selectivity factor of joining the tuples in $V_3 \wedge \overline{V_1}$ with those in V_4.

To compute the first term in (5), observe that $P(V_3, \overline{V_1}|R_1) = P(V_3|R_1) - P(V_3, V_1|R_1)$. We compute $P(V_3|R_1)$ using (1). To compute $P(V_3, V_1|R_1)$, we express it as $P(V_3, V_1|CQ_{V_3}, CQ_{V_1})P(CQ_{V_3}, CQ_{V_1}|R_1)$, which is then computed using (2).

Lastly, we consider how to derive (4). In general, the coverage $P(p, \overline{p_1}, \ldots, \overline{p_t}|Q)$ can be expressed as $\sum_{i=1}^{s} P(q_i|Q)$, where plans q_i can be derived using the procedure in Figure 5 and each $P(q_i|Q)$ can be computed in a manner similar to computing $P([V_3, \overline{V_1}]V_4|Q)$.

The key idea of this procedure is as follows: Given a bucket size n, the overlapping of sources in each bucket creates at most $2^n - 1$ disjoint source fragments. For example, the overlapping of sources in bucket $\{S, V\}$ creates at most 3 fragments $[S, V], [S, \overline{V}]$, and $[\overline{S}, V]$, where a fragment such as $[S, \overline{V}]$ contains all tuples in S but not in V. So given a query length m, the Cartesian product of source fragments in the buckets forms at most $(2^n - 1)^m$ disjoint plans. From the set of these plans we can find plans $q_i, i \in [1, s]$, that together ouput exactly the tuples in $p \wedge \overline{p_1} \wedge \ldots \wedge \overline{p_t}$.

Computing the coverage $P(p, \overline{p_1}, \ldots, \overline{p_t}|Q)$ (i.e., deriving plans q_i and computing $\sum_{i=1}^{s} P(q_i|Q)$) takes time $O(2^{(n-1)m})$. This is a substantial improvement over the time reported in (Florescu, Koller, & Levy 1997). In that framework, a formula such as $P(p, \overline{p_1}|Q)$ is expressed as $P(p|Q) - P(p, p_1|Q)$. In general $P(p, \overline{p_1}, \ldots, \overline{p_t}|Q)$ is expressed via 2^t terms and thus is computed in time $O(2^t)$. Since t refers to the number of plans involved, it can be as large as n^m.

Plan Coverage & Abstraction-Based Planning: In order for Streamer to work with plan coverage, we need to discuss how to add quantitative information to abstract sources, and how to use these information to compute the coverage of abstract plans.

Consider the concrete sources V_1 and V_2 in Figure 2.b, which are abstracted into V_{12}. We create abstract source V_{12} by grouping the statistics of V_1 and V_2 into *statistics intervals* and source descriptions of V_1 and V_2 into a *set of descriptions*. More specifically, for V_{12} (a) its source coverage is the interval $[\min_{i\in[1,2]} P(V_i|CQ_{V_i}), \max_{i\in[1,2]} P(V_i|CQ_{V_i})]$; (b) the number of distinct values of an attribute a is $[\min_{i\in[1,2]} d(a, V_i), \max_{i\in[1,2]} d(a, V_i)]$; and (c) its source description is the set $\{CQ_{V_1}, CQ_{V_2}\}$.

For example, if sources V_1 and V_2 have coverage 0.5 and 0.7, respectively; then we can assign the interval [0.5,0.7] to V_{12} as its coverage. Intuitively, this means that V_{12} represents concrete sources the coverage of which are between 0.5 and 0.7.

To compute the coverage of an abstract plan p, we proceed just like in the case of computing coverage of a concrete plan, but we use interval rather than point arithmetic.

More specifically, let V_* be an abstract source in plan p, and let CQ_* be the description of V_*. Then in computing the coverage of p, we need to compute quantities of the following forms: $P(V_*|CQ_*), \mathcal{F}(\cdots \bowtie (\cdots V_* \cdots) \bowtie \cdots)$, and $P(\ldots, CQ_*, \ldots|R)$. The first quantity is an interval given as a statistic on source V_*. The computation of the second quantity involves statistics on the number of distinct values of certain attributes in source V_*; these statistics are given as intervals, so we can use interval arithmetic to work with them. Finally, let CQ_* be the set $\{CQ_1, \ldots, CQ_l\}$,

[5]Since different plans typically involve different sources, $\mathcal{F}^1_V$ does vary between plans.

Input: Plans $p, p_1, p_2, \ldots, p_t$; query length m
Output: Plans $q_1, \ldots, q_s$
Let $S \leftarrow \emptyset$ and $x \leftarrow m$. **If** $x = 1$ **then** return plan $q_1 = p, \overline{p_1}, \ldots, \overline{p_t}$
else assume $p = V_1 \ldots V_m$ and $p_i = V_{i1} \ldots V_{im}, i \in [1, t]$.
Foreach source fragment $V_1, L_1, \ldots, L_t$, where L_i is either a positive V_{i1} or negative $\overline{V_{i1}}$ **do**

Assume that $L_1, \ldots, L_r$ are positive, and the rest are negative. Call CreatePlansQ on the input $p' = V2 \ldots V_m$, $p'_1 = V_{12} \ldots V_{1m}, \ldots, p'_r = V_{r2} \ldots V_{rm}$ with $m \leftarrow (x-1)$ to obtain plans $q'_1, \ldots, q'_u$. Add $[V_1, L_1, \ldots, L_t]$ to the beginning of each $q'_1, \ldots, q'_u$ to create plans $q_1, \ldots, q_u$. Add these plans to S.

Return S.

Figure 5: Algorithm for creating plans q_i

then we can compute $P(\ldots, CQ_*, \ldots|R)$ as $[\min_{i=1}^{l} P(\ldots, CQ_i, \ldots|R), \max_{i=1}^{l} P(\ldots, CQ_i \ldots|R)]$.

Experiments and Discussions

In this paper we focus on the evaluation of Streamer. We note that in a complete system, we also need to evaluate the actual coverage generated by Streamer plans. This evaluation is beyond the scope of this paper.

Strictly speaking, none of the previous related algorithms are directly comparable to Streamer. In particular, given a plan-coverage framework such as the one described in the previous section, the algorithms described in (Florescu, Koller, & Levy 1997) only return *approximate* plan orderings, whereas Streamer returns an *exact* plan ordering. Streamer also cannot be compared to existing algorithms that generate query-execution plans such as Sage (Knoblock 1995). Such algorithms solve a problem different from that of Streamer. They seek to find a complete execution plan (which in some cases cannot be decomposed at all into a set of conjunctive plans), whereas in a sense Streamer aims at finding an optimal ordering of "plan fragments", subjected to a certain optimization criterion.

We therefore opt to evaluate Streamer by comparing it with two brute-force algorithms that also compute the exact plan ordering. Our goal is to investigate to what extent Streamer is able to exploit source abstraction and plan independence information to speed up finding the exact plan ordering. BF is the best brute-force algorithm we could build, and is equivalent to Streamer without the abstraction component: to find the next best plan, it uses plan-independence information to decide the coverage of which plans may have changed and thus needs to be recomputed; it then compares concrete plans and outputs the one with the highest coverage. BFdrips is a brute-force version of Drips and is equivalent to Streamer without the independence-reasoning component: In each planning iteration it removes the previous best plan from the plan space, reabstracts the sources, then uses Drips to locate the next best plan.

We compared the three algorithms over a movie domain whose sources are randomly generated, and a Computer Science publication domain. In the publication domain we consider queries such as $Q(A_1, A_2, P)$:– co-author(A_1, A_2), paper(A_2, P) – find all papers written by co-authors. Sources with tuples satisfying relation co-author are constructed from researchers' webpages; sources with tuples satisfying relation paper are constructed from bibliographies available at http://liinwww.ira.uka.de/bibliography/.

For the results presented here, we use the following heuristic to abstract sources: For each bucket, first we classify sources into three categories: small, medium, and large, based on source coverage. Then for each category we iteratively perform random abstraction of its sources until we obtain a single abstract source. Finally we iteratively perform random abstraction on these categories' abstract sources until we obtain a single abstract source for the entire bucket.

Comparing Algorithms First we examine algorithm performance with respect to the size of the plan space. Many data-integration systems are interactive, and require that a plan-ordering algorithm produce the first plans fast. Figure 6.a shows the time to the first plan[6] of the three algorithms as we vary the bucket size in the movie domain. As expected, BF must evaluate all concrete plans, so its time grows exponentially[7]. The time of Streamer and BFdrips are orders of magnitude better than that of BF, and remain consistently linear below 10 msec (even when we sampled random bucket sizes from 6 to 20). Most likely this is because the time to the first plan of Streamer and BFdrips depends much less on the bucket size, but more critically on the quality of the abstraction; and the abstraction heuristics we use in Streamer and BFdrips were simple but fairly effective for finding the first plan. The same trends were also observed when we compared time to the 5th and 10th plan. These results show that our abstraction heuristic can reduce the time to first several plans from exponential to linear.

Figure 6.b shows the time to the last plan – the total planning time – of the three algorithms as we vary the bucket size. It is somewhat surprising that the time of BFdrips is the worst. A careful examination of BFdrips's behavior reveals the reason for this bad performance. As we previously noted, BFdrips outperforms BF in finding the first several plans. But as planning continues, the abstraction heuristic of grouping together sources of similar coverage gradually loses its usefulness: sources of similar size are no longer "similar" in contributing a roughly equal number of tuples to the answer. So BFdrips is now able to prune fewer and fewer abstract plans. Eventually, it will evaluate more abstract and concrete plans than it is able to prune, and the large time overhead incurred offsets its initial time gain. It is clear that BFdrips needs more sophisticated abstraction heuristics to substantially cut down on its total

[6]that is, time counting from the start of planning to the moment the algorithm outputs the first best plan

[7]Notice that time is plotted on a logarithmic scale against the bucket size.

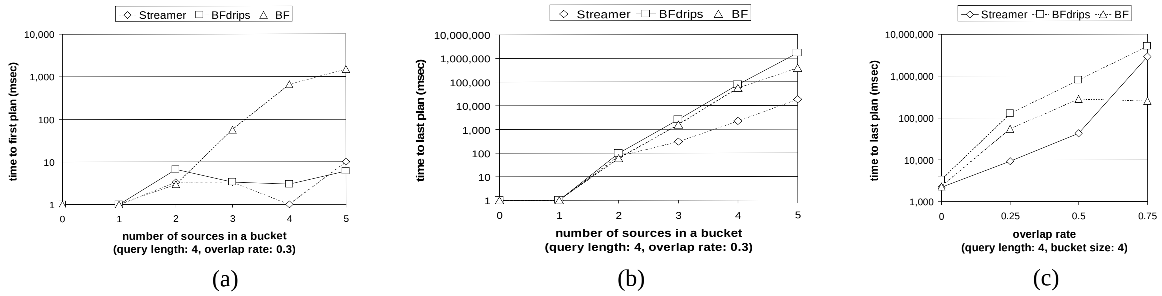

Figure 6: Run times of the three algorithms on the movie domain, as plotted on a logarithmic scale. The algorithms were implemented in Allegro Common Lisp 4.2; experiments were run on an SGI MIPS R4600 machine. Given a bucket size n, overlap rate r says that any source in the bucket overlaps with $n \cdot r$ other sources in that bucket.

planning time.

Figure 6.b also shows that Streamer outperforms BF by at least an order of magnitude, and that as the bucket size increases, the time gap between the two algorithms widens exponentially (as evidenced by the two roughly parallel time lines on the logarithmic scale). Streamer achieved improved performance because in each iteration it recomputed the coverage of far fewer plans than BF did. For example, in a comparison performed on a query, in each iteration BF recomputed the coverage of 87% of the plan space (on the average); in contrast, Streamer recomputed the coverage of only 32%: it "recycled" dominance relationships to eliminate 62% of the plan space, then used abstraction as in Drips to further reduce the remaining plan space by 6%.

Next we examine algorithm performance with respect to the degree of source overlap. Figure 6.c shows the time to the last plan of the three algorithms as we vary the overlap rate[8]. When there is no overlap at all (overlap rate 0), the hypothesis that Streamer outperforms BF, which in turn outperforms BFdrips, is supported by the results in Figure 6.c[9]. From this figure it also appears that Streamer consistently outperforms the other two algorithms as long as there are some non-overlapping sources. When there is full overlap (i.e., any two sources in a bucket overlap, which yields an overlap rate of 0.75 in this case), however, both Streamer and BFdrips perform substantially worse than BF. Streamer, for example, cannot exploit any plan-independence information, but still needs to maintain the dominance graph, thus incurring a large overhead as compared to BFdrips.

We ran additional experiments on both the movie and publication domains, in which we varied the query length, as well as the proportions of small, medium, and large sources in the buckets and the queries. We observed the same trends as have been discussed so far and shown in Figures 6.a-c.

Reducing Streamer's Exponential Complexity Finally we examine the question of making Streamer practical for large domains with hundreds of sources and large queries. Since Streamer's time to the last plan may grow exponentially (as is the case in Figure 6.b), Streamer may not scale for very large domains. We can improve scalability using a number of methods. First, we can use more sophisticated abstraction heuristics. The heuristic used in our experiments is very simple, but it reduced the time to the first several plans from exponential to linear, and reduced time to the last plan by an order of magnitude. A more sophisticated heuristic, such as one that takes into account plans that have been removed so far, could substantially reduce further total planning time. And second, we can slightly modify Streamer to make it into an anytime algorithm (Dean & Boddy 1988): in each iteration Streamer returns the best plan it has found so far when time allotted for that iteration is up; and the more time it has, the better the plan returned. Since in each iteration Streamer can quickly eliminate a large chunk of the candidate plan space by "recycling" the dominance relationships, the plan it returns is likely to be reasonably good. This modification to Streamer appears to be a good way of increasing scalability, but needs to be investigated more thoroughly.

[8]Overlap rate does not affect time to the first plan.

[9]The time at overlap rate 0 is 2220, 2290, and 3320 msec for Streamer, BF, and BFdrips, respectively.

Related Work

There are a variety of AI and database approaches to finding plans with high utility. The Markov Decision Process planning framework (Boutilier, Dean, & Hanks 1999) and traditional database query optimization (Selinger *et al.* 1979) exploit simple or highly regular utility models to build dynamic-programming-style planning algorithms. Dynamic programming techniques are not well suited for our plan-ordering problem due to the special structure of utility models such as plan coverage. Like Drips, the Sage planner (Knoblock 1995) also uses a branch-and-bound approach in which it iteratively builds, compares, and eliminates partial plans to find the optimal plan. It differs from Drips and Streamer in that partial plans are not constructed using abstraction and it considers simpler utility models. Planning by rewriting (Ambite & Knoblock 1997) finds approximately optimal plans by generating an initial solution plan, then rewriting the current solution plan in order to improve its quality using a set of declarative plan rewriting rules.

All of the planning approaches described above address a different problem from ours: they focus on finding a high-quality plan from a set of candidate plans, whereas we focus on achieving a high-quality plan by finding an efficient execution ordering of its fragments. Plan ordering was first suggested in (Florescu, Koller, & Levy 1997), which described approximate ordering algorithms that reason with a limited representation of plan coverage. We extended this coverage framework to conjunctive plans, and showed how reasoning about plan coverage can be incorporated into the ARP framework. Other works on reasoning about plan coverage and source overlap include (Friedman & Weld 1997; Lambrecht & Kambhampati 1998; Vassalos & Papakonstantinou 1998).

Conclusions and Future Work

We have described Streamer, which uses abstraction-based refinement planning and exploits information on plan independence to produce, in decreasing order of utility, a set of conjunctive plans that access data sources to obtain answers to user queries. The two key ideas here are about source abstraction and plan independence. The idea of reducing complexity by grouping sources with similar characteristics and reasoning with them as if with one single source can be applied to data-integration contexts in which we need to reason with a large number of sources. The idea of exploiting plan independence to reduce complexity is reminiscent of exploiting probabilistic independence to reduce the complexity of probabilistic inference – a well-known idea underlying Bayesian networks.

We have also shown how to use statistic information about the domain and data sources to estimate plan coverage, and how to incorporate the plan-coverage framework into Streamer. The key idea here is the recognition that plan coverage can be related back to the number of tuples returned by a plan, and that this number can be estimated using statistic information and estimation methods as employed in relational-database query optimization (Selinger *et al.* 1979).

We have presented preliminary experimental results suggesting that Streamer runs an order of magnitude faster than brute-force plan-ordering methods, which – as far as we know – are the only currently available methods to compute *exact* plan orderings. Finally, we have proposed methods to make Streamer scalable to large domains.

In the near future, we would like to examine more rigorously the scalability of Streamer, its anytime property, and how this property, as well as the choice of abstraction heuristics, can help in finding approximately optimal plan ordering for large domains.

Acknowledgment: We thank Adam Carlson, Steve Hanks, Zack Ives, Omid Madani, Rachel Pottinger, and the anonymous reviewers for valuable discussions and comments on this paper. This work is supported in part by ARPA/Rome Labs Grant F30602–95–1–0024, and in part by NSF grant IRI–9523649.

References

Ambite, J., and Knoblock, C. 1997. Planning by rewriting: Efficiently generating high-quality plans. In *AAAI '97*.

Boutilier, C.; Dean, T.; and Hanks, S. 1999. Planning under uncertainty: Structural assumptions and computational leverage. In *JAIR*. To appear.

Chawathe, S.; Garcia-Molina, H.; Hammer, J.; Ireland, K.; Papakonstantinou, Y.; Ullman, J.; and Widom, J. 1994. The tsimmis project: Integration of heterogenous information sources. In *IPSJ '94*. Japan.

Dean, T., and Boddy, M. 1988. An analysis of time-dependent planning. In *AAAI '88*, 49–54.

Florescu, D.; Koller, D.; and Levy, A. Y. 1997. Using probabilistic information in data integration. In *VLDB-97*, 216–225.

Friedman, M., and Weld, D. 1997. Efficiently executing information-gathering plans. In *IJCAI-97*, 785–791.

Haddawy, P.; Doan, A.; and Goodwin, R. 1995. Efficient decision theoretic planning: Techniques and empirical analysis. In *UAI-95*.

Ioannidis, Y. E. 1993. Universality of serial histograms. In *VLDB '93*.

Knoblock, C.; Minton, S.; Ambite, J.; Ashish, N.; Modi, P.; Muslea, I.; Philpot, A.; and Tejada, S. 1998. Modeling web sources for information integration. In *AAAI '98*.

Knoblock, C. 1995. Planning, executing, sensing, and replanning for information gathering. In *IJCAI '95*.

Lambrecht, E., and Kambhampati, S. 1998. Efficiently executing information gathering plan. In *AAAI '98 Workshop on AI and Information Integration*.

Levy, A. Y.; Rajaraman, A.; and Ordille, J. 1996. Querying heterogeneous information sources using source descriptions. In *VLDB-96*, 251–262.

Lipton, R., and Naughton, J. 1990. Query size estimation by adaptive sampling. In *PODS '90*, 40–46.

Mannino, M.; Chu, P.; and Sager, T. 1988. Statistical profile estimation in database systems. In *ACM Computing Surveys*, 20(3):191–221.

Ramakrishnan, R. 1998. Section 13.4: Estimating the cost of a plan. In *Database Management Systems*. McGraw-Hill.

Selinger, P.; Astrahan, M.; Chamberlin, D.; Lorie, R.; and Price, T. 1979. Access path selection in a relational database management system. In *SIGMOD '79*.

Vassalos, V., and Papakonstantinou, Y. 1998. Using knowledge of redundancy for query optimization in mediators. In *AAAI '98 Workshop on AI and Information Integration*.