

## Efficiently Ordering Query Plans for Data Integration

AnHai Doan & Alon Halevy

Department of Computer Science and Engineering

University of Washington, Seattle, WA, 98195

{anhai,alon}@cs.washington.edu

### Abstract

*The goal of a data integration system is to provide a uniform interface to a multitude of data sources. Given a user query formulated in this interface, the system translates it into a set of query plans. Each plan is a query formulated over the data sources, and specifies a way to access sources and combine data to answer the user query.*

*In practice, when the number of sources is large, a data-integration system must generate and execute many query plans with significantly varying utilities. Hence, it is crucial that the system finds the best plans efficiently and executes them first, to guarantee acceptable time to and the quality of the first answers. We describe efficient solutions to this problem. First, we formally define the problem of ordering query plans. Second, we identify several interesting structural properties of the problem and describe three ordering algorithms that exploit these properties. Finally, we describe experimental results that suggest guidance on which algorithms perform best under which conditions.*

### 1. Introduction

The goal of a data integration system is to provide a uniform interface to a multitude of data sources, thereby freeing the user from laborious manual interaction with the individual sources. The system provides this interface by allowing users to pose queries through a *mediated schema*, which is a virtual schema that captures the salient aspects of the application domain.

A data integration system typically consists of three main components: *query reformulator*, *optimizer*, and *execution engine*. Given a user query formulated in the mediated schema, the *query reformulator* translates it into a set of *query plans*. Each plan is a query formulated over the data sources, and specifies a way to access sources and combine data to answer the user query.

**Example 1.1** Consider a data integration system that answers queries related to movies. Suppose the system can access sources  $V_1, V_2, V_3$  that contain tuples  $\langle actor, movie \rangle$  and sources  $V_4, V_5, V_6$  that contain tuples  $\langle movie, review \rangle$ . Given a query asking for reviews of movies starring Harrison Ford, the reformulator may generate nine query plans, each accessing a source among  $V_1 - V_3$  to ask for the titles of movies starring Ford, then feeding these titles into a source among  $V_4 - V_6$  to obtain the reviews.  $\square$

Each query plan is then given to the *query optimizer*, which produces a *physical query execution plan*. A physical plan specifies exactly how the query plan is to be evaluated, including the order of the operations and the specific algorithm used for every operation (e.g., algorithms for joins, selections). Finally, the query execution plans are evaluated by the *query execution engine*. It is important to note that since sources may be incomplete, no single query plan is guaranteed to produce all the answers. Hence, the answer to a user query is the *union* of the output of *all* query plans.

Much research effort in data integration has concentrated on reformulation and optimization issues. Several algorithms to reformulate user queries have been proposed (e.g., [15, 5, 19]). Optimization is recognized as crucial to building practical data integration system, and hence has led to many works at all three levels of query evaluation: reformulation [4, 5, 7, 12], optimization [9, 23, 12], and execution [20, 11, 2].

At the reformulation level, most optimization approaches have focused on minimizing the cost to obtain *all* answers from the sources. In many data integration applications, however, the time to and the quality of the *first* answers is the most important. This is because for applications with a large number of sources, typically the number of query plans is very large and plan evaluation is costly, so executing all query plans is expensive and often infeasible. Furthermore, query plans tend to vary significantly in their utility (e.g., coverage, execution time, monetary cost, etc.), depending on which sources they access [13, 18].

Hence, an important optimization problem at the query-reformulation level is to find query plans in *decreasing order* of their utility, so that the data integration system can focus on and execute the best plans first. Query execution can then be aborted as soon as the user has found a satisfactory answer, or when allotted resource limits have been reached.

**Example 1.2** Consider *plan coverage*, defined as the number of tuples returned by a plan that haven't been returned by any plan executed previously [6, 7, 12]. If sources have equal access cost, then executing query plans in the decreasing order of their coverage returns as many answers as possible as soon as possible. Consequently, it maximizes the likelihood of obtaining a satisfactory answer early [6]. If sources have differing access cost, however, then prefer-

ences over coverage and cost can be modeled with the utility measure  $u(p) = \alpha * coverage(p) + \beta * cost(p)$ , where  $\alpha$  and  $\beta$  are constants specifying the tradeoffs [18]. Executing plans in the decreasing order of this utility value balances the desires of obtaining as many answers as possible with paying as little cost as possible.  $\square$

Several recent works have addressed the plan-ordering problem [6, 17, 13, 18, 22]. However, they dealt with restrictive data integration settings and considered only specific classes of utility measures. In this paper we address the general problem of plan ordering. We seek to modify the query-reformulation algorithm to output query plans in decreasing order of their utility, for a broad variety of utility functions. Since the time to execute plans is much higher than the time to find plans, we focus on finding the *first* few best plans. The rest of the plans can be found while the execution has begun.

Our contributions are as follows. First, we formally define the plan-ordering problem. Unlike most existing works, our definition does not assume any specific utility class. Second, we identify several interesting structural properties of the problem that provide insights into effective algorithms, then describe three plan-ordering algorithms that exploit these properties. Specifically, we describe a greedy algorithm that when applicable takes time linear wrt the number of sources to find the first several best plans, and two algorithms that use abstraction ideas from AI decision-theoretic planning. Finally, we describe experimental results that show that for several problem classes our abstraction-based algorithms can find the first several best plans very fast. The results also suggest guidance on which algorithms perform best under which conditions.

## 2. Background & Problem Definition

We now introduce a prototypical data integration architecture. Next, we discuss generating a set of query plans given a user query. Finally, we define the problem of ordering query plans.

**Mediated-Schema and Source Relations:** We model a data integration domain with a set 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. We model the contents of each data source with a *source relation*. We adopt the *local-as-view* approach [15, 12, 7] and describe each source relation 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 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  $\langle A, M \rangle$  such that actor  $A$  plays in the American movie  $M$ . Note that  $V_1$  may not necessarily contain *all* such tuples. Figure 1 lists the descriptions of six movie data sources.

<b>Schema relations:</b>	<b>Source relations:</b>
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)	$V_3(A,M) :- play-in(A,M)$
russian(M)	$V_4(R,M) :- review-of(R,M)$
	$V_5(R,M) :- review-of(R,M)$
	$V_6(R,M) :- review-of(R,M)$
<b>Sample query:</b>	
$Q(M,R) :- play-in(Ford,M), review-of(R,M)$	
<b>Bucket B1</b>	<b>Bucket B2</b>
$\{V_1, V_2, V_3\}$	$\{V_4, V_5, V_6\}$

**Figure 1. The data integration domain of movies.**

**User Queries and Conjunctive Query Plans:** A *user query* can be expressed as a conjunctive query  $Q(\bar{Y}) :- R_1(\bar{Y}_1), \dots, R_m(\bar{Y}_m)$ , where  $R_i$  are schema relations,  $\bar{Y}_i$  and  $\bar{Y}$  denote tuples of variables and constants, and  $\bar{Y} \subseteq \cup_{i=1}^m \bar{Y}_i$ . We refer to  $R_i(\bar{Y}_i)$  as the  $i$ -th *subgoal* of the query. For example, a query asking for all tuples  $\langle M, R \rangle$  such that  $R$  is a review of movie  $M$  starring Harrison Ford might be formulated as query  $Q$  shown in Figure 1.

A *conjunctive query plan*  $p$  (or *plan*  $p$  when there is no ambiguity) has the form  $p(\bar{Y}) :- V_1(\bar{U}_1), \dots, V_n(\bar{U}_n)$  where each  $V_i$  is a source relation corresponding to a data source, and the  $U_i$  denote tuples of variables and constants. The meaning of plan  $p$  is that it accesses and combines data from the sources  $V_1, \dots, V_n$ , to produce tuples  $\bar{Y}$  in response to query  $Q(\bar{Y})$ . For example,  $p :- V_1 V_4$  is a plan that answers query  $Q$  in Figure 1. A plan is *sound* if all answers it produces are answers to the user query. The union of the output tuples of *all* sound plans is the answer to the query.

**Generating Query Plans:** For ease of exposition, we shall use the *bucket algorithm* [16] to reformulate a user query into a set of sound query plans. We show in Section 7 how our plan-ordering algorithms can be adapted to work with other query-reformulation algorithms.

The bucket algorithm creates for each schema relation  $R$  (i.e., a subgoal) of the query a *bucket*, which is the set of all sources that can return tuples that satisfy  $R$ . For query  $Q$  in Figure 1, it creates two buckets  $B_1$  and  $B_2$  as shown in that figure. Next, the algorithm combines sources, one from each bucket, to form plans. Combining sources from buckets  $B_1$  and  $B_2$  forms nine plans:  $V_1 V_4, V_1 V_5, \dots, V_3 V_6$ . Finally, the algorithm tests each plan and outputs only the sound ones.

We now consider how to modify the bucket algorithm to output sound plans in *decreasing order* of their utility. Notice that, in practice, when the number of sources is large, there will be *many* query plans in the Cartesian product of the buckets. The bucket algorithm must test *each* of these plans for soundness. Hence, if we order plans *only after* all sound query plans have been generated, finding the first several best plans will be significantly delayed.

We can address this problem as follows. First we create

the buckets. Then we order plans in the Cartesian product of the buckets. If a plan coming out of the ordering algorithm is found to be sound, it is optimized for execution; otherwise, it is thrown away, and next plan is requested from the ordering algorithm. With this strategy we still execute all sound plans, and only sound plans, in the decreasing order of their utility. Furthermore, if sound plans are distributed uniformly over the plan ordering, then with a very high probability we shall find sound plans in the first several plans output by an ordering algorithm. For example, even when only 20% of plans are sound (which is unusually low), we still find a sound plan in the first 20 plans with probability  $1 - 0.8^{20} = 0.99$ . Hence, if we can find the first 20 plans fast, without examining *all* plans in the Cartesian product, then with a very high probability we would quickly find the first best sound plan.

In this paper we adopt the above strategy. So 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. Our focus is to find the first several plans in the ordering very fast, without generating all plans in the Cartesian product.

**Utility and the Plan-Ordering Problem:** The *utility* of a plan is commonly defined as a number that indicates the relative “worth” of the plan. Most existing works assume that the utility of a plan can be computed solely from the intrinsic properties of that plan. In many data integration settings, however, this utility value depends not only on the plan itself, but also on the plans previously executed and the user query (see the examples below). Hence, we adopt a general notion of utility, and define the *utility* of a plan  $p$  with respect to a set of plans  $\{p_1, \dots, p_l\}$  and a query  $Q$  to be a number indicating the relative “worth” of  $p$ , given that plans  $p_1, \dots, p_l$  have been executed. We denote this utility value as  $u(p|p_1, p_2, \dots, p_l, Q)$ .

**Example 2.1** Consider the total processing cost of a plan (e.g., execution time or monetary fee). When there is no caching, the cost of a plan is independent of any other plan. When caching is used, however, the cost of a plan may decrease (thus its utility increases) because a plan just executed has cached the results of some operations that this plan uses.

As another example, consider plan coverage [6, 7, 12, 21], which measures the number of “useful” tuples returned by a plan. Clearly, the coverage of a plan varies depending on which other plans have been executed and if they have accessed any source that overlaps sources accessed by this plan. Hence, following [6], we define *the coverage of a plan*  $p$  with respect to a set of plans  $\{p_1, \dots, p_n\}$  and a query  $Q(\bar{Y})$  to be the probability that a tuple chosen randomly among all tuples  $\bar{Y}$  that satisfy  $Q$  will be returned by plan  $p$  and not by any plan  $p_i, i \in [1, n]$ .  $\square$

We now define the plan-ordering problem as follows: given a set of query plans that answer a user query, find the best plan  $p_1$  (i.e., the one with the highest utility), then the next best plan  $p_2$ , assuming that  $p_1$  has been executed, then the next best plan  $p_3$ , assuming that  $p_1$  and  $p_2$  have been executed, and so on, up to plan  $p_k$ , for a given  $k$ . Formally,

**Definition 2.1 (Plan-Ordering Problem)** *Given a query  $Q$ , a utility measure  $u$ , and a number  $k$ , let  $S$  be the set of query plans generated in response to  $Q$ . Find the  $k$  plans with the highest utility in  $S$  in the decreasing order of their utility. That is, find an ordering  $\{p_1, \dots, p_k\}$ ,  $p_i \in S, i \in [1, k]$ , such that for each  $i \in [1, k]$ :*

$$u(p_i|p_1, \dots, p_{i-1}, Q) = \max_{p \in (S \setminus \{p_1, \dots, p_{i-1}\})} u(p|p_1, \dots, p_{i-1}, Q).$$

Section 7 reviews different variations of this definition, which have been addressed by the related work. The plan-ordering problem is challenging for two reasons. First, the utility of a plan may depend on the plans preceding it in the ordering. Hence, in these cases we simply cannot compute plan utilities *in isolation*, then sort the plans. Second, in the presence of a large number of sources, the set of plans is typically huge, making brute-force methods infeasible in all but the most trivial applications. Therefore, it is important to find key problem properties that enable us to design efficient solutions. In the next section we discuss four such problem properties.

### 3. Problem Properties

**Utility Monotonicity:** First, we discuss *utility monotonicity*, which when holds allows us to order plans by local comparisons of sources. Suppose we want to order plans in the Cartesian product of buckets  $B_1$  and  $B_2$  (Figure 1) in increasing order of their cost. Suppose also that the cost of any plan  $V_i V_j$  is  $cost(V_i V_j) = c_i + c_j$ , where accessing a source  $V_i$  ( $V_j$ ) incurs a constant cost  $c_i$  ( $c_j$ ). Then it is immediately obvious that given any plan, replacing a source in that plan with another source that has lower cost yields a better plan. Hence, there is a very effective greedy strategy to find the first best plan: find the source with the least cost from each bucket, then return the plan formed from the found sources.

This example motivates the notion of utility monotonicity. We say a utility function  $u$  is *monotonic* wrt a subgoal  $t$  of query  $Q$  iff (1) given any bucket  $B_t$  for this subgoal, we can find a total order  $\succeq$  on  $B_t$  such that for any two sources  $V_{ti} \succeq V_{tj}$  in  $B_t$ , replacing  $V_{tj}$  by  $V_{ti}$  in any plan yields a plan with higher utility, and (2) Property 1 holds regardless of the set of plans already executed. If  $u$  is monotonic wrt *all* subgoals of  $Q$  then we say it is *fully monotonic* wrt  $Q$ . The cost measure  $c_i + c_j$  is fully monotonic.

As another example, consider cost measure (from [23]):

$$cost(V_i V_j) = (h + \alpha_i * n_i) + (h + \alpha_j * n_j), \quad (1)$$

which applies if we retrieve all movies starring Ford from  $V_i$ , all tuples  $\langle M, R \rangle$  from  $V_j$ , then join them at the system site. Here  $h$  is the overhead cost of accessing a source,  $\alpha_i$  is the cost of transmitting a movie item from  $V_i$  to the system site,  $n_i$  is the number of items output by  $V_i$ ; and  $\alpha_j$  and  $n_j$  are analogous items for  $V_j$ .

This cost measure is fully monotonic because, intuitively, we can decrease  $\text{cost}(V_i V_j)$  by decreasing term  $\alpha_i * n_i$  independently of term  $\alpha_j * n_j$ , and vice versa. Clearly, any cost measure that is a linear combination of independent terms, where each term represents the cost of a constituent source, is fully monotonic.

Now consider cost measure (also from [23]):

$$\text{cost}(V_i V_j) = (h + \alpha_i * n_i) + (h + \alpha_j * (n_j * n_i / N)), \quad (2)$$

which applies if we retrieve all movies starring Ford from  $V_i$ , then perform the join between them and tuples  $\langle M, R \rangle$  at  $V_j$ . Here the parameters are defined as in (1), and  $N$  is the total number of movies across the sources. The term  $(n_j * n_i / N)$  is an estimation of the number of items output by  $V_j$ . This cost measure is monotonic wrt the second subgoal, but not to the first subgoal. However, if transmission costs  $\alpha$  are the same across all sources, then it is also monotonic wrt the first subgoal, and thus is fully monotonic.

The above examples show that there are many practical utility measures that are fully monotonic. For these utility measures, we can apply algorithm **Greedy** described in Section 4 to efficiently order plans. However, there are also many utility measures that are not fully monotonic, such as plan coverage and the utility measures described in Section 6. To efficiently order plans for such utility measures, we exploit different problem properties, which we describe next.

**Source Similarity:** We say two sources are *similar* if replacing one source by another in *any* plan changes the utility of that plan very little. Large data integration domains tend to have many similar sources. For example, consider the domain of digital cameras. The hundreds of online sources that sell digital cameras can be naturally divided into several groups. At one end, we have small resellers that offer cameras at steep discount prices, but have poor customer service. We also have small, specialized stores that deal exclusively with cameras; they charge higher prices, but have excellent customer service. At the other end of the spectrum, we have large national chains that sell electronic goods, such as *Best Buy* and *Circuit City*. They carry extensive offerings of products, with reasonable customer service and average to high prices. In between, we have stores such as *Target*, *Wal-Mart*, and *Costco*, which also provide reasonable customer service, but do not offer high-end cameras. Besides resellers, there are at least 30–40 sites that review digital cameras (see *consumersearch.com*). These sites can also be naturally divided into several groups, such

as free sites (e.g., *dpreview.com*), and sites that charge a fee (e.g., *consumerreports.org*).

The presence of many similar sources makes large data integration domains especially suited to abstraction techniques based on source similarity. Similar sources can be grouped and reasoned with as with a single source. For example, suppose the user wants to buy a high-end camera and greatly values good customer service. Then by examining the characteristics of the reseller groups, a data integration system will know that it should first consider large national electronic chains, or small, specialized stores. It knows to exclude the other groups without having to examine each individual reseller in that group, thus obtaining substantial computational savings. In Section 5.2 we introduce **iDrips** and **Streamer**, two algorithms that exploit such techniques to order plans.

#### **Plan Independence and Utility-Diminishing Returns:**

Two additional properties that prove especially useful are *plan independence* and *utility-diminishing returns*. The **Streamer** algorithm exploits these properties in conjunction with abstraction to efficiently order plans.

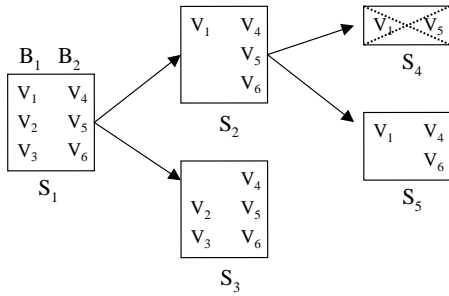
Two plans are said to be *independent* of each other iff the utility of a plan does not depend on whether the other has been executed. Often, there exists an efficient and sound (but perhaps not complete) procedure to infer the independence of a plan pair simply by inspecting their constituent sources. This is important because some algorithms we introduce later rely on such procedures to obtain plan independence information efficiently. For example, in the case of plan coverage, we can show that two plans are independent of each other if there are at least two corresponding constituent sources that do not overlap and contribute some attribute values to the output tuples of the plans.

*Utility-diminishing returns* refers to the property that the utility of a plan does not increase if it is “pushed” further down the plan ordering. This property holds for plan coverage because the number of *new* tuples returned by a plan can not increase as we execute more plans.

## 4. The Greedy Algorithm

This algorithm applies when the utility measure is fully monotonic. We now explain its working via a simple example. We use the term *plan space* to refer to the set of plans formed by taking the Cartesian product of a set of buckets. In Figure 2, buckets  $B_1 = \{V_1, V_2, V_3\}$  and  $B_2 = \{V_4, V_5, V_6\}$  form a plan space  $S_1$ . Suppose we want to apply **Greedy** to  $S_1$ . Since full monotonicity holds, **Greedy** finds the best plan using a simple strategy: finds the best source in each bucket, then returns the plan formed from the found sources. Let this plan be  $V_1 V_5$ .

**Greedy** now proceeds to find the second best plan. First, it removes the best plan  $V_1 V_5$  from plan space  $S_1$ . As we show shortly, this removal splits  $S_1$  into the set of plan



**Figure 2. Removing plan  $V_1V_5$  from the plan space  $S_1$  results in two plan spaces  $S_3$  and  $S_5$ .**

spaces  $\{S_3, S_5\}$  that together contain all plans in  $S_1$  except plan  $V_1V_5$ . Next, **Greedy** finds the best plan for  $S_3$  and the best plan for  $S_5$ , using the same greedy strategy described above. Finally, it compares these two best plans and returns the one with the highest utility as the second best plan. It proceeds in a similar manner to find subsequent best plans.

We now explain how **Greedy** removes plan  $V_1V_5$  from plan space  $S_1$ . The basic idea is recursive splitting starting with the first bucket. **Greedy** splits the first bucket into two buckets:  $\{V_1\}$  and  $\{V_2, V_3\}$ . This splits  $S_1$  into two smaller plan spaces  $S_2$  and  $S_3$ . Next, it focuses on  $S_2$ , because  $S_2$  contains plan  $V_1V_5$ . **Greedy** splits the second bucket of  $S_2$  into two buckets:  $\{V_5\}$  and  $\{V_4, V_6\}$ , thus splitting  $S_2$  into two plan spaces  $S_4$  and  $S_5$ . Finally, it removes  $S_4$ , which contains exactly  $V_1V_5$ . The end result of removing  $V_1V_5$  from  $S_1$  is then the two plan spaces  $S_3$  and  $S_5$ .

In [3] we formally describe **Greedy** and prove that, given a fully monotonic utility measure, **Greedy** returns the correct order of the first  $k$  best plans, where  $k$  is a pre-specified threshold. We also prove that **Greedy** runs in  $O(mn^2k^2)$  time, where  $m$  is the largest bucket size and  $n$  is the query length.

## 5. Abstraction-Based Algorithms

We now consider the case in which utility monotonicity does not hold. Since large data integration domains tend to have many similar sources, as we have argued in Section 3, we consider the use of similarity-based abstraction. First, we introduce **Drips**, a planning algorithm that uses abstraction to find the best plan from a set of plans [10]. Then we describe **iDrips** and **Streamer**, two algorithms we have developed that extend **Drips** to order plans.

### 5.1. Drips: An Abstraction-Based Planner

The basic idea of **Drips** 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 (query) 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 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.

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 3.a and 3.d. Here sources are represented with circles, the overlaps of which mean overlaps of the actual sources.

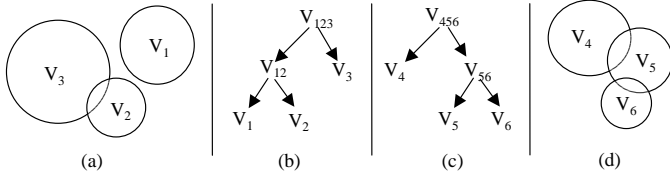
**Drips** proceeds in two stages. In the first stage, it iteratively groups and abstracts sources in each bucket. For the bucket in Figure 3.a, it abstracts two sources  $V_1$  and  $V_2$  into the abstract source  $V_{12}$ , which is then abstracted with  $V_3$  to yield the final abstract source  $V_{123}$  for this bucket, as shown in Figure 3.b. A similar abstraction process is shown in Figure 3.c for the second bucket. The abstract plan  $V_{123}V_{456}$  represents all nine concrete plans  $\{V_1, V_2, V_3\} \times \{V_4, V_5, V_6\}$ .

In the second stage, **Drips** iteratively refines, evaluates, and eliminates abstract plans until it finds the best plan. It starts 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. Let's say **Drips** picks  $V_{123}$  and refines it into the two component sources  $V_{12}$  and  $V_3$ , yielding two plans  $V_{12}V_{456}$  and  $V_3V_{456}$ .

Assume **Drips** computes the coverage of these two plans to be  $[0.1, 0.7]$  and  $[0.5, 0.8]$ , respectively. Since these intervals overlap, **Drips** cannot eliminate either plan. It then picks a plan, say  $V_{12}V_{456}$ , and refines 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 **Drips** can eliminate  $V_2V_{456}$  because this plan is dominated by  $V_1V_{456}$ .

We now have two plans left:  $V_3V_{456}$  and  $V_1V_{456}$ . **Drips** picks  $V_3V_{456}$  and refines 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 **Drips** can eliminate both dominated plans. Since **Drips** now has only plan  $V_3V_4$  left and it is a concrete plan, **Drips** returns  $V_3V_4$  as the plan with the highest coverage, and as the first plan in the ordering.

Note that in total we need to compute the coverage of only six plans to find the best plan out of nine. This represents a saving of 33% over the brute-force approach in terms of the number of plans evaluated. Even though the six plans evaluated include abstract plans, evaluating an abstract plan is just slightly more expensive than evaluating a concrete plan, because the former can be carried out just like the latter, but with interval rather than point arithmetic; see



**Figure 3. Concrete and abstract sources in two buckets for a query in the movie domain.**

Appendix A.3 of [3] for more details. Furthermore, source abstraction in Drips can be done efficiently using a variety of heuristics [10].

### 5.2. Abstraction-Based Plan Ordering

Drips is not suited for data integration because it finds only the first plan in the ordering. We have developed two algorithms that extend Drips to find subsequent plans.

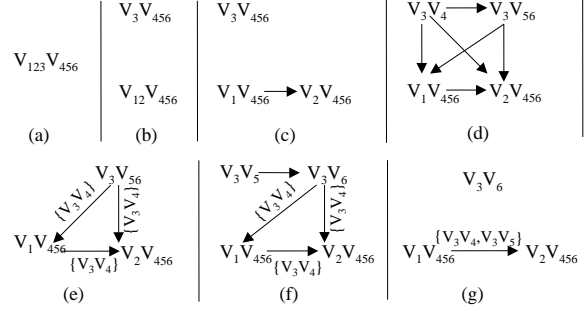
**The iDrips Algorithm:** This algorithm is a straightforward extension of Drips. It begins by applying Drips to find the first best plan, then removes that plan from the original plan space. This removal results in a set of new plan spaces that together contain all plans from the original plan space, except for the first best plan. In the next iteration, iDrips reabstracts the sources in the new plan spaces, then applies Drips to find the second best plan, and so on.

In each iteration, while comparing plans iDrips implicitly establishes many dominance relations of the form “plan  $p$  is better than plan  $q$ ”, which it then discards at the end of that iteration. In the next iteration iDrips completely rebuilds the abstract plan space and may reestablish the same dominance relations. Thus in finding the next best plan, iDrips duplicates a lot of the work done in previous iterations. This observation led to the development of the second abstraction-based algorithm, Streamer, which is a more sophisticated extension of Drips.

**The Streamer Algorithm:** This algorithm is applicable if utility-diminishing returns holds. It abstracts sources only once, at the beginning. It then exploits plan independence and diminishing-returns properties to keep track of dominance relations among abstract plans, and reuses these relations in the search for the next best plans.

The Streamer algorithm is shown in Figure 5, and we shall explain its working on the example of Section 5.1.

**Finding the Best Plan & Creating Dominance Relations:** Initially, after abstracting the sources, 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 4.a and 4.b, and Step 2.c of Figure 5). 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 4.c and Step 2.b of Figure 5), denoted as  $V_1V_{456} \rightarrow V_2V_{456}$ . We associate each



**Figure 4. Stages of the domination graph created by Streamer on the example in Figure 3. The set associated with a link is shown next to that link.**

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 4. 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, Streamer picks  $V_3V_{456}$  and refines it into  $V_3V_4$  and  $V_3V_{56}$  (Figure 4.d and Step 2.c of Figure 5). 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).

**Removing the Best Plan:** 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 4.e and line 1 of Step 2.d of Figure 5). 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 ( $V_3$  and  $V_2$  overlap in Figure 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.

**Finding the Second Best Plan & Recycling Dominance Relations:** After removing  $V_3V_4$ , Streamer needs to recheck the validity of each remaining link. To do this Streamer uses information on plan independence. It 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

<p><b>Input:</b> Query <math>Q</math>, utility measure <math>u</math>, set of buckets <math>B_1, \dots, B_m</math>, and a threshold <math>k</math>; <b>Output:</b> The best <math>k</math> plans in <math>B_1 \times \dots \times B_m</math>, in decreasing order of utility</p> <ol style="list-style-type: none"> <li><b>1. Foreach</b> bucket <math>B_i</math> <b>do</b> create top abstract source <math>T_i</math>. Put top plan <math>P = T_1 \dots T_m</math> into a dominance graph <math>G</math>. Set <math>u(P) \leftarrow nil</math>.</li> <li><b>2. Loop</b> until the best <math>k</math> plans have been returned: <ol style="list-style-type: none"> <li><b>(a) Foreach</b> nondominated plan <math>a</math> in <math>G</math> such that <math>u(a)</math> is <i>nil</i> <b>do</b> recompute <math>u(a)</math>.</li> <li><b>(b) Foreach</b> pair of nondominated plans <math>b, c \in G</math> such that <math>u(b) = [l_b, h_b]</math>, <math>u(c) = [l_c, h_c]</math>, and <math>l_b \geq h_c</math> <b>do</b> create link <math>b \rightarrow c</math> and set <math>E(b, c) \leftarrow \emptyset</math>.</li> <li><b>(c) If</b> exist abstract, nondominated plans in <math>G</math> <b>then</b>  Pick one such plan, say <math>p</math>; refine <math>p</math> into <math>p_1, \dots, p_l</math>; set <math>u(p_i) \leftarrow nil</math>; and add <math>\{p_1, \dots, p_l\}</math> to <math>G</math>.  <b>Foreach</b> plan <math>p' \in G</math> st <math>p \rightarrow p'</math>, and for each plan <math>p_i, i \in [1, l]</math>, st <math>CheckValidity(p_i, E(p, p')) = true</math> <b>do</b> create <math>p_i \rightarrow p'</math>, and set <math>E(p_i, p') \leftarrow E(p, p')</math>.  Remove plan <math>p</math> from <math>G</math> and go back to Step 2.a.</li> <li><b>(d) If</b> there are no abstract nondominated plans in <math>G</math> <b>then</b> pick any nondominated plan <math>d \in G</math> and output <math>d</math> as the next best plan. Remove <math>d</math> from <math>G</math>.  <b>Foreach</b> link <math>q \rightarrow q'</math> in <math>G</math> <b>if</b> <math>CheckValidity(q, E(q, q') \cup \{d\}) = true</math> <b>then</b> add <math>d</math> to <math>E(q, q')</math> <b>else</b> remove <math>q \rightarrow q'</math> from <math>G</math>.  <b>Foreach</b> plan <math>e \in G</math> such that <math>e</math> and <math>d</math> are not independent <b>do</b> set <math>u(e) \leftarrow nil</math>.</li> </ol> </li> </ol> <p><b>Algorithm</b> <math>CheckValidity(\text{plan } r, \text{ set of concrete plans } F)</math>: Return <i>true</i> iff there exists a concrete plan <math>r' \in r</math> such that <math>r'</math> is independent of all concrete plans <math>f \in F</math>.</p>
--

**Figure 5. A description of Streamer**

$E(p, q)$  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 because the utility model provides diminishing returns<sup>1</sup>.

Returning to our example, we can easily check that after removing  $V_3V_4$  all three links shown in Figure 4.e are still valid. For example, link  $V_3V_5 \rightarrow V_1V_{456}$  is valid because plan  $V_3V_6 \in V_3V_5$  is independent of plan  $V_3V_4$  in  $E(V_3V_5, V_1V_{456})$ . 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 5), **Streamer** picks a nondominated plan, say  $V_3V_5$  in Figure 4.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 4.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 4.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.

In [10] the authors show that **Drips** always terminates, returning the best plan. This result can be used to prove that when utility-diminishing returns holds, **Streamer** returns the correct order of the first  $k$  best plans, for a given  $k$  value.

## 6. Empirical Evaluation

We have performed experiments to evaluate **iDrips** and **Streamer**. We did not consider **Greedy** because it clearly outperforms the other algorithms when applicable. Both **iDrips** and **Streamer** return the correct plan ordering, so we evaluate them with respect to running time. Our goals were to examine which algorithm is appropriate under which

<sup>1</sup>For example, after removing a plan  $r$ , the coverage of any plan  $t$  remains unchanged if  $t$  is independent of  $r$ , and decreases otherwise.

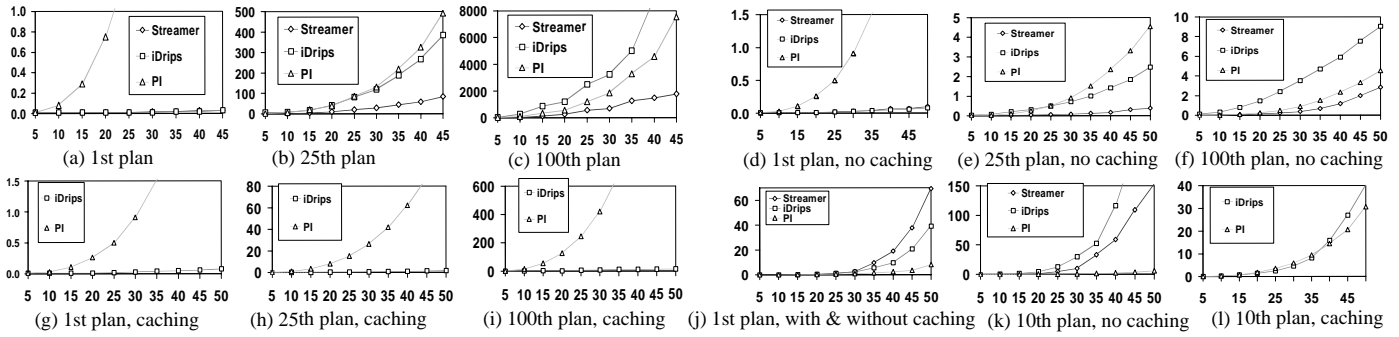
conditions, and to show that there are practical utility measures that are not fully monotonic, for which **iDrips** and **Streamer** can quickly find the first several best plans.

Since none of the related algorithms in the literature is directly comparable with our algorithms (see Section 7), we compared them with **PI**, the best brute-force algorithm that also computes the exact plan ordering. **PI** serves as a reference point for our experiments. In each iteration, **PI** uses plan independence information to decide the utility of which plans may have changed and thus need to be recomputed.

We experimented with four utility measures for which full monotonicity does not hold. The first one is plan coverage, which was mentioned in Section 2, and is described in detail in Appendix A of [3]. The second one is the cost measure (2) in Section 3, where transmission costs vary across sources. The third one is the same as (2), except that now accessing a source may fail with a probability. The fourth one is the average monetary cost per output tuple:  $u(p) = Cost(p)/NumOutputTuples(p)$ , where  $Cost(p)$  is computed using (2), and  $NumOutputTuples(p)$  is computed as in [23]. For the experiments we use synthetic data and a simple abstraction heuristic that groups sources based on their similarity wrt the number of expected output tuples. A detailed description of the utility measures, the data, and the abstraction heuristic can be found in [3].

The graphs in Figure 6 plot the time it takes from when the query is issued until the the first  $k$  best plan have been found, against the bucket size, for various utility measures. We do not count the time it takes to generate the buckets, because this step can be done efficiently and takes the same time for all three algorithms. Experiments were run on a Pentium III 500 Dell Dimension with 128 MB RAM and Linux Red Hat 5.2. We now discuss experimental results for each utility measure in turn.

**Plan Coverage:** Figures 6.a-c show the followings. First, **Streamer** performs very well compared to **PI** in finding the first several plans. This is because the abstraction heuristic **Streamer** uses is very effective in finding the first plan: across all runs the number of plans evaluated by **Streamer** in the first iteration is less than 4% of the number of plans evaluated by **PI**. Furthermore, since the



**Figure 6. Experimental results with query length 3 for various utility measures: (a)-(c) plan coverage, (d)-(i) cost with source failure, and (j)-(l) average monetary cost.**

abstraction heuristic is effective, the size of the dominance graph of **Streamer** increases only slowly as **Streamer** outputs plans. Hence the overhead of **Streamer** (in maintaining dominance graph, abstracting sources, and refining and eliminating plans) also increases only slowly.

Second, **iDrips** also achieves good performance in finding the first several best plans, but not as good as **Streamer**. This is because **Streamer** was able to recycle many dominance relations. Consequently, in each iteration it reevaluated far fewer plans than **iDrips** did.

And finally, **iDrips** performs worse than **PI** at finding the 100-th plan (Figure 6.c). This is because as the number of output plans increases, the abstraction heuristic of grouping together sources of similar amount of output tuples gradually loses its usefulness: sources with roughly the same amount of output tuples are no longer “similar” in contributing a roughly equal number of *new tuples* to the answer. As the abstraction heuristic becomes less accurate, **iDrips** is able to prune fewer abstract plans. Eventually, it ends up evaluating more plans than **PI**.

The results in Figures 6.a-c are obtained at the overlap rate 0.3, that is, each source in a bucket overlaps with 30% of other sources in the bucket. We experimented with other overlap rate and observed the same trends reported above. We also observed that **Streamer**’s relative performance compared to **PI** in finding subsequent plans decreases as the degree of plan independence decreases (i.e., as the overlap rate increases). This is because as the overlap rate increases, it invalidates more and more dominance relations, so **Streamer** can recycle fewer and fewer dominance relations. Consequently, **Streamer**’s runtime increases substantially, whereas **PI**’s runtime increases at a lower rate.

**Cost Measure (2) & Cost with Probability of Source Failure:** Experimental results for these two cost measures are very similar, so we report only the results for cost with probability of source failure. We ran experiments with both the no-caching and caching options to evaluate the algorithms at different degrees of plan independence.

Figures 6.d-f show the results for the no-caching case. Here, full plan independence and thus utility-diminishing

returns hold. Hence, in addition to **iDrips** and **PI**, **Streamer** is also applicable. The results show that **Streamer** performs substantially better than **iDrips** and **PI**, and finds the first several plans very fast. Again, as in the case of plan coverage, **Streamer**’s abstraction heuristic is very effective in finding the first plan, and it is able to recycle many dominance relations in subsequent iterations.

Figures 6.g-i show the result when we cache the results of source operations. The cost of a subsequent source operation is set to zero if its result has been cached. Consequently, there is some plan dependence in the domain: a plan is independent of another plan only if they do not share any source operation. Furthermore, here utility-diminishing returns does not hold, and hence **Streamer** is not applicable. The results show that **iDrips** performs very well compared to **PI** and finds the first several plans very fast. This is because **iDrips**’s abstraction heuristic remains fairly effective even after a substantial number of plans has been found, so **iDrips** evaluates very few plans in each iteration as compared to **PI**.

**Average Monetary Cost per Tuple:** Figures 6.j-l show the results for both no-caching and caching cases. The graphs show that both **Streamer** and **iDrips** perform worse than **PI** in finding the first several plans. Here, the abstraction heuristic is not as effective as the ones in previous utility cases, so **Streamer** and **iDrips** still evaluate fewer plans than **PI**, but not much fewer. Therefore the computational gain in plan evaluation is small. Furthermore, since the overhead is roughly proportional to the number of plans evaluated, and the number of plans evaluated is large, the overhead is also large, thus offsetting the gains.

We also experimented with different domain & source parameters, and abstraction heuristics, but observed the same trends discussed above. We also experimented with varying query length from 1 to 7, and observed the same trends, but with increasing performance gaps as the query length increases.

**Summary:** From the experiments, we can draw the following conclusions. First, for several problem classes **Streamer** and **iDrips** substantially outperformed **PI** and

found the first several plans very fast, without generating the complete set of sound plans.

Second, performance of **Streamer** and **iDrips** depends on the tradeoff between the number of plans evaluated and the overhead of maintaining dominance graph and refining and eliminating plans. In general, they are appropriate when the domain is amenable to abstraction and an effective abstraction heuristic is used, so that they evaluate few plans and incur little overhead.

Third, **iDrips** performs best when it can find an efficient abstraction heuristic, not only for finding the first plan, but also for subsequent plans.

Finally, **Streamer** performs best when it is difficult to find an efficient heuristic for **iDrips** (e.g., in the case of plan coverage), and the degree of plan dependence is relatively small, so that **Streamer** can recycle many dominance relations.

## 7. Discussion & Related Work

We now address several limitations of our approach and discuss related work. First, throughout the paper we have used the *bucket* algorithm to generate plans, we now show how the plan-ordering algorithms can be adapted to handle other plan-generation methods. Consider the *inverse-rule* algorithm [5]. This algorithm creates *inverse rules*, which specify for each schema relation all different ways to obtain tuples from the sources. The algorithm then forms a datalog plan by adding the rules to the original query. Executing the datalog program produces all the answers for the query. When user queries are conjunctive (as is the case considered in this paper), the inverse rules that cover the same schema relation naturally form a bucket. Hence, we can apply our plan-ordering algorithms to these buckets, much in the same way we apply them in conjunction with the bucket algorithm.

The case of recursive datalog plans [12] is not handled by our current framework, due to its lack of a treatment for recursive plans. However, we note that [8] has extended **Drips** to successfully handle recursive plans. This work thus serves as a good starting point on future research to extend **iDrips** and **Streamer** to the recursive case.

Another recently proposed plan-generation algorithm is *minicon* [19]. This algorithm creates *MCDs*, each of which in effect refers to a source that covers a *set* of query subgoals. It then combines the *MCDs* to form plans: a set of *MCDs* that together cover *all* subgoals forms a sound plan. We can easily modify this algorithm to work with our ordering algorithms as follows. First, we change it to produce *generalized buckets*, where each bucket covers a *set* of subgoals (as opposed to a *single* subgoal in the bucket algorithm). Then we create plan spaces, each of which is a set of buckets that together cover all subgoals. Finally, we apply the plan-ordering algorithms to the plan spaces. Changes to

*minicon* are simple, and modifying the ordering algorithms to handle a set of plan spaces (instead of one) is trivial. It is important to note that the plan spaces of *minicon* contain only sound plans. Hence, here we do not have to test plans output by the ordering algorithm for their soundness, as in the case with the bucket algorithm.

**Query Optimization:** Research on query optimization for data integration fall into three groups.

*Works at Query-Reformulation Level:* Several works [4, 7, 12] translate the user query into a query plan, then use local completeness assertions [14] to remove redundant parts of the query plan. The query plan in these works corresponds to the union of our query plans. So they optimize the cost to get *all* answers from the sources. In contrast, by ordering query plans we focus on optimizing the cost to the *first* answers.

Ordering query plans was proposed in [15], and was subsequently investigated in [6, 17, 13, 18, 22]. In [6] the authors consider a restricted data integration setting where each query plan accesses a *single* source. They described several algorithms that were designed to work specifically with source coverage. These algorithms output *approximate* plan orderings, whereas ours output *exact* orderings.

In [17, 13] the authors address the same plan-ordering problem as ours. In [13] they develop a branch-and-bound algorithm to significantly speed up plan ordering. However, this algorithm differs from ours in two important aspects. First, it does not deal with the case when the utility of a plan depends on the plans already executed (i.e., it assumes full plan independence). Second, it is designed to return all *k* plans *at once*, for a given *k*. It is not clear if the algorithm can be modified to output plans incrementally. Hence, these works are not directly comparable to ours. In [18] the authors address a related but different problem. They seek to find the best *parallel* query plan, one that allows accessing multiple sources for each query subgoal. Their utility measure is a linear combination of execution time and (the log of) plan coverage. The linear-combination nature of the utility measure allows them to design an efficient System-R style algorithm to find the best parallel plan. In [22] the authors seek to find the best parallel plan that maximizes plan coverage, under a given cost limit. An interesting twist in this work is that they also consider the case of intermittent source unavailability.

*Works at Query-Optimization Level:* In [12], the authors present a method for ordering the access to sources to reduce the execution cost. In [9] the **Garlic** system finds the query execution plan with the least cost using a cost-based optimizer for traditional databases. In [23] the authors discuss generating efficient execution query plans for fusion queries, a subclass of the data integration problem. In [1] the authors propose generating an initial solution plan, then

iteratively rewriting the current solution in order to improve it. All these works focus on minimizing the cost to get *all* answers, rather than the cost to the *first* answers.

*Works at Query-Execution Level:* During execution even the best plan (i.e., the first one in the ordering) may still turn out to be inefficient due to unexpected network delay or inaccurate source statistics. To address such issues, several works [20, 11, 2] propose techniques on adapting query execution plans and interleaving planning and execution. These works are similar to ours in that they also aim to minimize time to the first answers. However, they perform such optimization at the query-execution level, whereas ours works at the query-reformulation level.

## 8. Conclusions

Data integration systems play an important role in helping users obtain information efficiently from a multitude of data sources. In the presence of large number of sources, however, data integration systems must process a huge number of query plans. The query plans are costly to evaluate, and vary significantly in their utility. Hence, it is crucial that such systems generate the best plans quickly and execute them first, in order to guarantee acceptable time to and the quality of the first answers.

In this paper we have made several contributions toward solving this problem. First, we provided a formal definition of the problem as ordering query plans in decreasing order of their utility. Unlike most existing works, our problem definition does not assume any particular utility class and does not assume complete plan independence. Second, we identified four interesting problem properties and developed three plan-ordering algorithms that exploit these properties. Our theoretical and experimental results show that for a variety of problem classes, our algorithms find the best query plans very quickly. The results also suggest guidance on which algorithms performing best under which conditions.

**Acknowledgment:** We thank Adam Carlson, Steve Hanks, Zack Ives, Omid Madani, Rachel Pottinger, and the anonymous reviewers for valuable discussions and comments on earlier drafts of this paper. This work is supported in part by ARPA/Rome Labs Grant F30602-95-1-0024, NSF grant IRI-9523649, and NSF grant IIS-9978567.

## References

- [1] J. Ambite and C. Knoblock. Flexible and scalable query planning in distributed and heterogeneous environments. In *Proc. of the 4th Int. Conf. on AI Planning Systems (AIPS)*, 1998.
- [2] R. Avnur and J. Hellerstein. Continuous query optimization. In *SIGMOD '00*, 2000.
- [3] A. Doan and A. Y. Halevy. Efficiently ordering query plans. Technical report, Department of Computer Science and Engineering, University of Washington, 2001.
- [4] O. Duschka. Query optimization using local completeness. In *Proc. of the 14th Nat. Conf. on AI*, 1997.
- [5] O. Duschka and M. Genesereth. Answering recursive queries using views. In *Proc. of PODS '97*.
- [6] D. Florescu, D. Koller, and A. Y. Levy. Using probabilistic information in data integration. In *Proc. of VLDB '97*.
- [7] M. Friedman and D. Weld. Efficiently executing information-gathering plans. In *Proc. of the Int. Joint Conf. of AI (IJCAI)*, 1997.
- [8] R. Goodwin. Using loops in decision-theoretic refinement planners. In *Proc. of the 3rd Int. Conf. on AI Planning Systems (AIPS)*. AAAI Press, 1996.
- [9] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proc. of VLDB '97*, 1997.
- [10] P. Haddawy, A. Doan, and R. Goodwin. Efficient decision theoretic planning: Techniques and empirical analysis. In *Proc. of the Nat. Conf. on Uncertainty in AI (UAI)*, 1995.
- [11] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld. An adaptive query execution system for data integration. In *Proc. of SIGMOD*, 1999.
- [12] E. Lambrecht, S. Kambhampati, and S. Gnanaprakasam. Optimizing recursive information gathering plans. In *Proc. of the Int. Joint Conf. on AI (IJCAI)*, 1999.
- [13] U. Leser and F. Naumann. Query planning with information quality bounds. In *Proc. of the Int. Conf. on Flexible Query Answering Systems (FQAS)*, 2000.
- [14] A. Y. Levy. Combining artificial intelligence and databases for data integration. 1999. To appear in a special issue of LNAI: Artificial Intelligence Today; Recent Trends and Developments.
- [15] A. Y. Levy, A. Rajaraman, and J. Ordille. Query-answering algorithms for information agents. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1996.
- [16] A. Y. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. of VLDB*, 1996.
- [17] F. Naumann, U. Leser, and J. C. Freytag. Quality-driven integration of heterogeneous information systems. In *Proc. of VLDB '99*.
- [18] Z. Nie and S. Kambhampati. Joint optimization of cost and coverage of information gathering plans. Technical report, Dept. of CSE, Arizona State Univ., 2001. <http://rakaposhi.eas.asu.edu/ParPlan.pdf>.
- [19] R. Pottinger and A. Y. Levy. A scalable algorithm for answering queries using views. In *Proc. of VLDB '00*.
- [20] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost based query scrambling for initial delays. In *Proc. of SIGMOD '98*.
- [21] V. Vassalos and Y. Papakonstantinou. Using knowledge of redundancy for query optimization in mediators. In *Proceedings of the AAAI Workshop on AI and Information Integration*, 1998.
- [22] R. Yerneni, F. Naumann, and H. Garcia-Molina. Maximizing coverage of mediated web queries. Technical Report [www-db.stanford.edu/yerneni/pubs/mcmwq.ps](http://www-db.stanford.edu/yerneni/pubs/mcmwq.ps), Stanford University, 2000.
- [23] R. Yerneni, Y. Papakonstantinou, and H. Garcia-Molina. Fusion queries over internet databases. In *Proc. of the 6th Int. Conf. on Extending Database Technology (EDBT)*, 1998.