Fusion Queries over Internet Databases

Serge Abiteboul, Hector Garcia-Molina, Yannis Papakonstantinou, Ramana Yerneni
Department of Computer Science
Stanford University
{abiteboul, hector, yannis, yerneni}@cs.stanford.edu

Abstract

Fusion queries search for information integrated from distributed, autonomous sources over the Internet. In this context, data is not cleanly fragmented as in traditional distributed databases, and the number of sources participating in a typical query is large. We investigate techniques for efficient processing of fusion queries. First, we focus on a very wide class of query plans that capture the spirit of many techniques usually considered in existing systems. We show how to efficiently find, under various realistic scenarios, good query plans within this large class. We evaluate the performance of these plans and provide additional heuristics that, by considering plans outside our target class of plans, yield further performance improvements.

1 Introduction

In distributed information systems on the Internet, one often encounters sources that provide incomplete and possibly overlapping information on a set of entities. A *fusion query* searches over these entities, looking for ones that satisfy given conditions. To illustrate, consider databases operated by the Departments of Motor Vehicles (DMVs) of several states. Conceptually, each state database can be thought of as a relation R_i with the following attributes, among others:

- Driver's license number (L): includes state and number within that state.
- Violation (V): A code representing an infraction, for example "driving under the influence" (*dui*) or "speeding" (*sp*).
- Date (D): the date of the infraction.

Figure 1 shows some sample relations for three DMVs. Now, consider a fusion query that searches for drivers who have both a *dui* and a *sp* violation. For instance, the driver with license *J55* satisfies this query because he has a *dui* infraction in the first state and a *sp* one in the second state. License *T21* also meets this criterion, but the others in Figure 1 do not. Notice that the information for a particular driver, say *J55*, may be dispersed among the sources, so the query conceptually (but not actually) "fuses" the information for each driver as it checks the constraints. This means that such a query is inherently expensive: we do not know in advance which sources may have the information fragments for *J55*, so the search must cover all possibilities.

To express our sample query in SQL, we first let U be the union of all the R_1, R_2, ... tables at the various DMVs, and then we write

$$
\begin{array}{ll}
\texttt{SELECT} & u_1.L \\
\texttt{FROM} & U\ u_1, U\ u_2 \\
\texttt{WHERE} & u_1.L = u_2.L \\
 & \texttt{AND}\ \ u_1.V = sp\ \texttt{AND}\ u_2.V = dui
\end{array}
$$

R_1		
L	V	D
J55	dui	1993
T21	sp	1994
T80	dui	1993

R_2		
L	V	D
T21	dui	1996
J55	sp	1996
T11	sp	1993

R_3		
L	V	D
T21	sp	1993
S07	sp	1996
S07	sp	1993

Figure 1: DMV Example.

This is the type of query we focus on in this paper.

We believe that such fusion queries are important now, and will become even more important in the future as integration systems cope with more and more information that has *not* been nicely structured and partitioned in advance. In a traditional distributed database environment, for instance, an administrator could determine in advance that all violations for licenses issued in a given state go to a particular database. This makes query processing much simpler because we only have to perform the query locally at each database and union the results. However, in a world where sites and databases are autonomous, it is very difficult to agree on and enforce global partitions. For example, the California DMV may want to keep a record of all violations that occurred in its state regardless of the license's origin. At the same time, the California DMV may not have complete records for California drivers because the other DMVs may not notify the California DMV of infractions occurring in their states involving California drivers.

We see this pattern occurring over and over: A phone company keeps three customer databases, managed by different organizations (billing, service & repairs, planning) and each with overlapping and differing information. A state library system has databases at different locations, and there are multiple records for the same book, each with slightly different bibliographic information. The World Wide Web introduces even more fusion scenarios: the same products are described by different catalogs on the Web, the same books are offered by different on-line bookstores, and so on. In all these cases there is a strong need to query this autonomous information as if it were an integrated collection, to be able to discover opportunities (e.g., the same book is offered at the lowest price at one bookstore) or to have accurate information (e.g., we want to have all the facts about a customer to decide if we mail the customer promotional material).

Not only are fusion queries very important in Internet databases but they also introduce tough performance challenges. Even the best of query execution plans will require a lot of work, since they have to check all the possible ways in which an entity can be found to match the selection conditions based on information from various sources. The amount of required work grows rapidly as the number of sources or the number of conditions to check grows. So, if we do not have good execution plans, it will be simply infeasible to answer nontrivial queries. Hence, the role of the query optimizer is very critical for fusion queries. Furthermore, the optimizer itself needs to be efficient as there are huge numbers of possible plans that can be considered.

Traditional query optimizers do not consider fusion queries in any special way, and hence may produce poor plans. They typically assume that the number of sources is small, and rely on good fragmentation of data which as we have argued is not the case with Internet databases. Our goal in this paper is to understand how fusion queries can be processed efficiently in the context of Internet databases. To gain this knowledge, we proceed in four steps:

1. To make our task more manageable, we only consider fusion queries that retrieve the merge attribute (e.g., driver's license) of the matching entities. If additional information on the matching entities (e.g., their address) is needed, a "second phase" query would have to be issued. We believe this is a reasonable simplification for a first paper on fusion query optimization. The assumption eliminates some performance factors (e.g., when should additional

attributes be fetched), but still lets us study the basic types of fusion query plans.

Furthermore, the "two-phase" approach is sometimes used in practice so it is interesting in its own right. For instance, in a bibliographic search scenario, one first identifies the documents that satisfy the criteria, and then fetches the documents, usually a few at a time. The main reason why searches are split this way is that the full records of the matching entities may be very large and are often stored on separate systems altogether. For example, a driver's name, address, and physical features may be stored separately from the violations information. Even when this is not the case, this two-phase processing may reduce cost because we do not pay the price of fetching full records until we know which ones are needed.

2. To understand the types of fusion query plans, we first narrow down the space of plans to those we call *simple plans*. Simple plans are coordinated by a central site we call the *mediator*. The mediator can ask one or more sources to evaluate a condition, obtaining a set of values for the merge attributes. The mediator can also perform one or more semijoins by sending a set of merge attribute values to a source and receiving that subset whose elements match a condition at the source.[1] Finally, the mediator can combine the sets of merge attribute values it obtains via union or intersection operations. For our sample query, one simple plan, call it $\mathcal{P}_1$, could be as follows: First, the mediator asks each source to give it all L values for drivers with $V = dui$ (see Figure 1). Then the mediator unions all these values, and sends this set to all sources, asking each to select from those the ones that have $V = sp$. The union of those answers would be the final answer.

   The class of simple plans is very large; it includes all the strategies that most real world optimizers would currently develop for a fusion query, plus many other natural plans. Thus, even though we have narrowed down our search, we expect to still find some excellent plans in this space, at least as good as those found by current optimizers.

3. The class of simple plans is still too large to be searched in a brute force way for an optimal plan. Fortunately, using theorems we will prove here, we can constrain our search to a much smaller class of plans, those we call *semijoin-adaptive*. Intuitively, these are simple plans that work on the conditions of the query in some order, one condition at a time. (Plan $\mathcal{P}_1$ above is also semijoin-adaptive, since it first considers one condition fully, and then moves on to the second one.) It turns out that semijoin-adaptive plans are very good under a general cost model that we use here. In particular, if there are only two query conditions, or if there are more conditions but they are independent, then the best semijoin-adaptive plan is also the best simple plan. In this case the optimizer can perform a significantly smaller search over all semijoin-adaptive plans, and still find the best simple plan. Even if the conditions of the query are not independent, we argue that the best semijoin-adaptive plan provides an excellent heuristic. Indeed, when dealing with autonomous sources over the Internet, we often have no information about the dependence of conditions, so using the best semijoin-adaptive plan is as good a guess as we can make.

4. Once we find the best semijoin-adaptive plan, we consider some variations of this plan that make it non-simple but that may improve performance further. In other words, as a "postoptimization" step, we consider a class of plans that is larger than simple plans, but we only perform a local optimization in the neighborhood of our best semijoin-adaptive plan. For example, one of the variations we consider is having the mediator use set difference. To illustrate, consider our sample plan $\mathcal{P}_1$. We leave the first part unchanged, obtaining the set $X_1 = \{J55, T80, T21\}$ of all L values that have $V = dui$ at some source. Now, instead of

[1] Note that if the source does not directly support semijoins, the mediator can emulate them; see Section 2.3.

sending X_1 to all sources for a semijoin, we only send it to the first source. The first source (R_1) returns the subset of X_1 with $V = sp$, i.e., $Y_1 = \{T21\}$. Now the mediator knows that $T21$ is definitely an answer to the query, so when it goes to the second source, it does not have to send the full X_1 set; instead it sends $X_1 - Y_1$, reducing the amount of data that must be sent to the source. In Section 5 we discuss this and other postoptimization techniques.

In summary, the contributions of this paper are as follows:

- We define and study the evaluation of fusion queries (Section 2). As far as we know, ours is the first paper that studies in depth the optimization of such queries.
- We define important classes of plans for fusion queries, including simple and semijoin-adaptive plans, and other subclasses of them (Section 2.5).
- Using a cost model that factors in both communication costs and query processing costs at the sources, we present an optimization algorithm for efficiently finding the optimal semijoin-adaptive plan (Section 3). The algorithm runs in time linear in the number of sources participating in the fusion. This is obviously a very important property when fusing information from many sources.
- We prove that in many commonly occurring situations, the optimal semijoin-adaptive plan is also the optimal plan under the broader class of simple plans (Section 4). This lets us find the optimal simple plan with the same, efficient algorithm for semijoin-adaptive plans.
- We describe a set of postoptimization heuristics that can improve the performance of the optimal semijoin-adaptive plan further (Section 5).
- We present (Section 6) an evaluation of the fusion plans generated using the various techniques. This allows us to quantify the gains achievable in some scenarios, and to identify important trends.
- We discuss (Section 7) how the optimization strategies we have developed here can be incorporated into existing general purpose optimizers in order to achieve efficient fusion query processing.

2 Framework

In this section, we provide the framework for fusion query optimization. First, we define the operations and data exported by sources. Second, we formally define fusion queries. Then we describe the class of simple plans, and our cost model. Finally, we identify some important subsets of simple plans.

2.1 The Sources

In our framework, each source has a *wrapper* [31] that exports a relation[2]. All the source relations have the same attributes, which include the merge attribute M. Attribute M identifies the real-world entity that the tuple refers to. Internally, each source can use a different model, but the wrapper maps it to the common view we are using. Note that we use a relational framework here only for simplicity. The algorithms we propose in this paper can be extended in a straightforward way to other data models. Incidentally, our interest in the fusion problem emerged from the TSIMMIS project which uses a semistructured object model [30].

[2]The wrapper can export other relations of course; here we focus on the one involved in the fusion query of interest.

We assume here that all wrappers can indeed provide a common merge attribute M, such as the driver's license in our DMV example. In most cases of interest, there is such an attribute, but it may be represented in slightly different ways at each source. The differences can range from simple ones, e.g., one source uses hyphens between the state and the number in a license and another does not, to complex ones, e.g., customers are identified only by name, and names can be misspelled. The most practical way to cope with these problems is to develop a unique *canonical form* for the merge attribute (and hopefully for the rest of the attributes too). Then the wrapper or the mediator uses mapping functions [29] to translate data from the source into the canonical form and translate queries with canonical values into queries that use source specific values. Unfortunately, canonical forms are not always applicable (e.g., consider the case of misspellings.) However, to deal with a more tractable problem, in this paper we assume that canonical forms are available and used.

In our framework, wrappers support the following two types of operations:

- *Selection queries* denoted as $X := sq(c_i, R_j)$. This operation retrieves the set of items that satisfy c_i in source relation R_j (we use the term *item* to refer to a merge attribute value). We leave unspecified for now, on purpose, what exactly constitutes a valid condition except to say that it is something that all wrappers should be able to process. On the Internet, for example, sources typically offer very limited capabilities, so conditions may be of the form attribute equals constant. For more sophisticated sources, conditions could be any first order predicate involving R_j attributes.

- *Semijoin queries* denoted as $X := sjq(c_i, R_j, Y)$. This operation computes the subset of Y items that satisfy c_i in R_j. There are sources that may not provide a semijoin operation directly. It is possible in such cases that the wrappers for such sources provide the semijoin capability. And in those cases where the wrappers also do not provide the semijoin operation, the mediator can "emulate" the semijoin by issuing individual selection queries, one for each value in Y. The cost of the emulated operation will be higher than if the wrapper supported the semijoin operation, and this will be taken into account in our cost model. Our assumption does imply that wrappers are at least able to handle selection conditions of the form c_i AND $M = m$, where m is a "passed binding" (from Y) and c_i is a condition in the fusion query.

2.2 Fusion Queries

We use U to refer to the union of all the source relations R_j. The general form of fusion queries is:

$$\begin{array}{ll} \texttt{SELECT} & u_1.M \\ \texttt{FROM} & U\ u_1, \ldots, U\ u_m \\ \texttt{WHERE} & u_1.M = \ldots = u_m.M \\ & \texttt{AND}\ \ c_1\ \texttt{AND}\ \ldots\ \texttt{AND}\ c_m \end{array}$$

where each condition c_i, $i = 1, \ldots, m$ involves only one u_i variable and U attributes, and is supported by the wrappers[3]. Note that the query only retrieves the merge attribute M, as discussed in Section 1.

[3]Actually, each u_i can refer to a different union view U_i which represents the union of all the source relations R_j that pertain to condition c_i. For simplicity of exposition we consider that all the U_is are the same. The entire discussion of this paper and the techniques developed here are directly applicable to the case where the U_is may differ.

2.3 Simple Plans

Under *simple plans*, mediators can issue selection and semijoin queries to the wrappers, and can themselves perform operations of the form $X := Y \text{ } op \text{ } Z$, where Y and Z are sets of items, and op is either a union ($\cup$) or an intersection ($\cap$). Notice that, in general, mediators would have to perform joins; however, in our framework for fusion queries, wrappers only return sets of items (unary relations), so joins are equivalent to intersections. A simple plan is thus a sequence of selection queries, semijoin queries, and local operations. Figures 2(a), 2(b) and 2(c), later on in this section, give examples of simple plans for a fusion query with 3 conditions and 2 sources.

Simple plans are quite general and can represent many ways to efficiently execute fusion queries. They allow the description of any plan obtained by standard algebraic optimization techniques such as pushing selection and projection operations to the sources. They also allow query rewriting using the distributivity of join and union, the commutativity and associativity of join and union, along with many techniques to reorder joins efficiently to process m-way joins, and the use of semijoin operations for efficient processing of joins in distributed environments. These are strategies that most optimizers typically use. Thus, if we are able to find the best simple plan, we believe we will have a plan that cannot be beaten by existing real-world optimizers. (There are, however, techniques not covered by simple plans, and some of them will be considered in Section 5.)

Simple plans can be employed in many contexts with a wide range of source capabilities. Even if some of the sources do not support semijoin queries, the mediator can emulate a semijoin query as a set of selection queries. Another approach to dealing with sources that do not support semijoin queries is to assign an infinite cost to that operation. All query plans that use the unsupported semijoin queries will be automatically eliminated by the optimizer (because of the infinite cost assigned to such plans).

2.4 Cost Model

In our domain of interest, the Internet databases, the most time consuming task is sending queries to the sources and receiving answers from them. Thus, we adopt a model that emphasizes these costs and neglects the cost of local processing at the mediator. In particular,

- Each $sq(c_i, R_j)$ and each $sjq(c_i, R_j, X)$ operation has a non-negative cost.
- If X, Y and Z are sets of items with $X = Y \cup Z$, the cost of $sjq(c_i, R_j, X)$ is at most as much as the sum of the costs of $sjq(c_i, R_j, Y)$ and $sjq(c_i, R_j, Z)$ for any c_i and R_j. In other words, there is no benefit in splitting a semijoin set X into semijoin sets Y and Z.
- The cost of local mediator operations, $\cup$ and $\cap$, is negligible.
- The cost of a query plan is the sum of the costs of the constituent $sq(c_i, R_j)$ and $sjq(c_i, R_j, X)$ operations. Thus, our focus is on total work involved in query execution, not on response time. (We briefly discuss response time issues in Section 8.)

We do not make any assumptions as to how the costs of source queries are computed; they could take into account the cost of communicating with sources, and the cost of actually processing the queries at the sources. The costs can vary depending on the contents of R_j and X, and the selectivity of c_i.

We believe that our cost model is quite general. Many distributed database optimizers use cost models that are compatible with our cost model. In fact, our cost model allows for cost estimation that can deal with heterogeneous source characteristics while many other cost models do not.

1)	$X_{11} := sq(c_1, R_1)$	1)	$X_{11} := sq(c_1, R_1)$	1)	$X_{11} := sq(c_1, R_1)$
2)	$X_{12} := sq(c_1, R_2)$	2)	$X_{12} := sq(c_1, R_2)$	2)	$X_{12} := sq(c_1, R_2)$
3)	$X_1 := X_{11} \cup X_{12}$	3)	$X_1 := X_{11} \cup X_{12}$	3)	$X_1 := X_{11} \cup X_{12}$
4)	$X_{21} := sq(c_2, R_1)$	4)	$X_{21} := sjq(c_2, R_1, X_1)$	4)	$X_{21} := sjq(c_2, R_1, X_1)$
5)	$X_{22} := sq(c_2, R_2)$	5)	$X_{22} := sjq(c_2, R_2, X_1)$	5)	$X_{22} := sq(c_2, R_2)$
6)	$X_2 := X_{21} \cup X_{22}$	6)	$X_2 := X_{21} \cup X_{22}$	6)	$X_2 := X_{21} \cup X_{22}$
7)	$X_2 := X_2 \cap X_1$	7)	$X_{31} := sq(c_3, R_1)$	7)	$X_2 := X_2 \cap X_1$
8)	$X_{31} := sq(c_3, R_1)$	8)	$X_{32} := sq(c_3, R_2)$	8)	$X_{31} := sq(c_3, R_1)$
9)	$X_{32} := sq(c_3, R_2)$	9)	$X_3 := X_{31} \cup X_{32}$	9)	$X_{32} := sq(c_3, R_2)$
10)	$X_3 := X_{31} \cup X_{32}$	10)	$X_3 := X_2 \cap X_3$	10)	$X_3 := X_{31} \cup X_{32}$
11)	$X_3 := X_3 \cap X_2$			11)	$X_3 := X_2 \cap X_3$
(a)	A filter plan	(b)	A semijoin plan	(c)	A semijoin-adaptive plan

Figure 2: Three simple plans

2.5 Important Classes of Simple Plans

To conclude this section, we define some important classes of simple plans for fusion queries.

1. *Filter plans*: Filter plans are simple plans that use only selection queries and local operations at the mediator. Many traditional distributed query optimizers do not use semijoin operations ([35, 15]). Such optimizers generate filter plans for fusion queries.

   Figure 2(a) shows an example filter plan for a fusion query with conditions c_1, c_2, and c_3 and sources R_1 and R_2. In this plan, the mediator pushes each condition to each source (six selection queries), and computes the final answer from the corresponding item sets. In general, every filter plan issues mn selection queries, where m is the number of conditions and n is the number of sources, and performs local computations at the mediator. Of course, the order of these operations can be different from that in Figure 2(a).

2. *Semijoin plans*: These are simple plans that employ semijoin queries in a restricted fashion. A particular semijoin plan is determined first by an ordering, say, $c_1, ..., c_m$ of the query conditions. The set X_1 of items satisfying c_1 at some source is first retrieved by issuing selection queries, one for each of the n sources. Next, the plan can either evaluate the second condition in a similar fashion, or it can evaluate it by semijoin queries using X_1 as semijoin set. In either case, the plan computes X_2, the set of items that satisfy c_1 at one source and c_2 at another (possibly the same) source. The process continues in a similar fashion for the rest of the conditions. In general, for a given ordering of the conditions, a particular semijoin plan is specified by deciding for each condition c_i (i in $[2..m]$), whether to evaluate c_i by selection queries or by semijoin queries using as semijoin set the set of items satisfying $c_1 \wedge ... \wedge c_{i-1}$.

   Figure 2(b) illustrates a semijoin plan for the same fusion query used in Figure 2(a). The second condition is evaluated by semijoin queries, and the others by selection queries. Observe that the first condition in a semijoin plan is always evaluated by selection queries, as there is no semijoin set to start with.

   Semijoin plans can be more efficient than filter plans. For instance, in Figure 2(b), the source with R_1 only returns a fraction of the items satisfying c_2, while the equivalent filter plan would have fetched all items in R_1 satisfying c_2. However, we can make semijoin plans even more effective by allowing them more flexibility. In particular, notice that for a given condition, semijoin plans either send selection queries to all sources or they send semijoin queries to all sources. This may be inefficient in an environment where the sources have widely different characteristics. For example, if the second source does not directly support semijoins (i.e.,

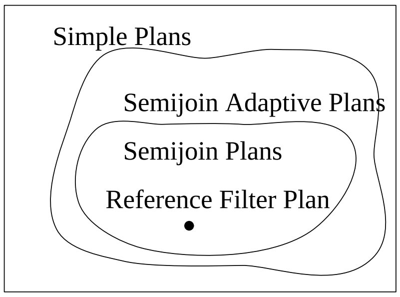


Figure 3: Relation between the subsets of simple plans

semijoins have to be emulated in an expensive manner) it may not be beneficial to process c_2 at the second source using semijoins. The class of plans described next has the ability to adapt to the characteristics of the sources.

3. *Semijoin-adaptive plans*: Like semijoin plans, these plans process one condition at a time, in some order $c_1, ..., c_m$. However, for each condition in $[2..m]$ and each source, the plan can choose independently between a selection query or a semijoin query. Figure 2(c) illustrates a semijoin-adaptive plan for the same sample query of the previous figures. As we can see, the plan processes c_2 at R_1 by issuing a semijoin query, and at R_2 by the use of a selection query. Thus, the plan can use the best strategy at each source.

It is important to note that semijoin plans are a special case of semijoin-adaptive plans, which in turn, are a special case of simple plans. This is illustrated by Figure 3. Filter plans are *not* a subset of semijoin-adaptive plans, since they are not restricted to compute the sets X_1, $X_2, ...$ used in the definition above. (They are of course not a superset since they do not use semijoins.) However, under our cost model, all filter plans have the same cost, i.e., the cost of performing the mn selection queries. Neither the order in which the selection queries are performed in a filter plan nor the mediator operations executed changes this cost. Therefore, for query optimization purposes, we can pick a single *reference* filter plan within the space of filter plans, like the one in Figure 2(a), and ignore the rest. Since this reference filter plan also lies in the space of semijoin plans, we can *conceptually* think of filter plans as a subset of semijoin plans.

In Section 4, we will prove that in many cases the optimal semijoin-adaptive plan is also optimal within the space of all simple plans. Furthermore, in Section 6, we compare the performance of our plans and demonstrate that semijoin-adaptive plans have a significant performance advantage over filter and semijoin plans. However, before that, we discuss how to find optimal plans within the plan space of Figure 3.

3 Finding Optimal Simple Plans

In this section we present three algorithms — FILTER, SJ, and SJA — that compute the best filter, semijoin, and semijoin-adaptive plans respectively. These algorithms are very efficient, as they run in time linear in the number of sources participating in the fusion query.

The algorithms use the following:

1. A function $sq\_cost(c_i, R_j)$ that estimates the cost of the selection query $sq(c_i, R_j)$.

2. A function $sjq\_cost(c_i, R_j, X)$ that estimates the cost of the semijoin query $sjq(c_i, R_j, X)$.

These functions can use whatever information is available at query optimization time, e.g., the expected size of data sets at the sources, the selectivities of the conditions at the various sources,

INPUT: Conditions $c_1, \ldots, c_m$
Sources $R_1, \ldots, R_n$
Cost functions *sq_cost* and *sjq_cost*
OUTPUT: An optimal semijoin plan
METHOD:

```
Optimal_Plan_Cost ← ∞
for every ordering [c_{o_1}, ..., c_{o_m}] of the conditions                          loop A
   Plan ← [X_{11} := sq(c_{o_1}, R_1), ..., X_{1n} := sq(c_{o_1}, R_n), X_1 := ∪_{j=1,...,n} X_{1j}]
   Plan_Cost ← Σ_{j=1,...,n} sq_cost(c_{o_1}, R_j)
   for i = 2, ..., m                                                                   loop B
      selection_queries_cost ← Σ_{j=1,...,n} sq_cost(c_{o_i}, R_j)
      semijoin_queries_cost ← Σ_{j=1,...,n} sjq_cost(c_{o_i}, R_j, X_{i-1})
      if selection_queries_cost < semijoin_queries_cost
         append toPlan the sequence
            [X_{i1} := sq(c_{o_i}, R_1), ..., X_{in} := sq(c_{o_i}, R_n), X_i := X_{i-1} ∩ (∪_{j=1,...,n} X_{ij})]
         Plan_Cost ← Plan_Cost + selection_queries_cost
      else
         append to Plan the sequence
            [X_{i1} := sjq(c_{o_i}, R_1, X_{i-1}), ..., X_{in} := sjq(c_{o_i}, R_n, X_{i-1}), X_i := ∪_{j=1,...,n} X_{ij}]
         Plan_Cost ← Plan_Cost + semijoin_queries_cost
   if Plan_Cost < Optimal_Plan_Cost
      Optimal_Plan ← Plan
      Optimal_Plan_Cost ← Plan_Cost
```

Figure 4: The SJ algorithm

and the capabilities of the sources (i.e., whether they can support semijoin operations directly or not). (For an illustration of particular cost estimates please see Section 6.) In analyzing the complexity of the various algorithms presented in this section, we assume that *sq_cost* and *sjq_cost* take constant time per invocation.

The FILTER algorithm: FILTER directly outputs the single reference filter plan (see Section 2.5). Hence FILTER does not spend any time searching for the optimal plan. Its running time is proportional to the number of steps required for producing the reference filter plan which, in turn, is proportional to the number of statements in the reference filter plan. Hence, the complexity of FILTER is $O(mn)$, where m is the number of conditions and n is the number of sources.

The SJ algorithm: SJ (see Figure 4) generates all possible $m!$ orderings of the conditions (see loop A). For each one of them, it generates the best *Plan* with respect to this ordering, estimates its cost, and eventually selects as *Optimal Plan* the one with the least cost among all orderings.

The best *Plan* with respect to a specific ordering $[c_{o_1}, \ldots, c_{o_m}]$ starts with a sequence of operations that evaluate c_{o_1} using selection queries. Then SJ goes over each one of the $m-1$ conditions c_{o_i}, $i = 2, \ldots, m$ (see loop B) and decides whether c_{o_i} is evaluated by semijoin or selection queries. In particular, SJ sums up and compares the cost of the n selection queries against the cost of the n semijoin queries. The *Plan* and its cost are appropriately updated in each round. The complexity of SJ is $O((m!)mn)$ because loop A iterates $m!$ times, loop B iterates $m-1$ times, and the operations inside loop B are of complexity $O(n)$.

The SJA algorithm: SJA (see Figure 5) differs from the SJ algorithm in that it makes a separate

```
INPUT: Conditions c_1, ..., c_m
         Sources R_1, ..., R_n
         Cost functions sq_cost and sjq_cost
OUTPUT: The optimal semijoin-adaptive plan
METHOD:
         Optimal_Plan_Cost ← ∞
         for every ordering [c_{o_1}, ..., c_{o_m}] of the conditions                      loop A
            Plan ← [X_11 := sq(c_{o_1}, R_1), ..., X_1n := sq(c_{o_1}, R_n), X_1 := ∪_{j=1,...,n} X_1j]
            Plan_Cost ← Σ_{j=1,...,n} sq_cost(c_{o_1}, R_j)
            for i = 2, ..., m                                                              loop B
               for j = 1, ..., n                                                           source loop
                  if sq_cost(c_{o_i}, R_j) < sjq_cost(c_{o_i}, R_j, X_{i-1})
                     append to Plan the operation X_ij := sq(c_{o_i}, R_j)
                     Plan_Cost ← Plan_Cost + sq_cost(c_{o_i}, R_j)
                  else
                     append to Plan the operation X_ij := sjq(c_{o_i}, R_j, X_{i-1})
                     Plan_Cost ← Plan_Cost + sjq_cost(c_{o_i}, R_j, X_{i-1})
               append to Plan the operation X_i := X_{i-1} ∩ (∪_{j=1,...,n} X_ij)
            if Plan_Cost < Optimal_Plan_Cost
               Optimal_Plan ← Plan
               Optimal_Plan_Cost ← Plan_Cost
```

Figure 5: The SJA algorithm

decision between selection and semijoin query for each condition at each source. In particular, SJA includes the "source loop" of Figure 5 where, for a given condition c_{o_i}, it decides, for each source R_j, whether the processing of c_{o_i} at R_j will be done with a semijoin query or a selection query.

It is easy to see that SJA's complexity is also $O((m!)mn)$ since the source loop iterates n times and the operations inside it cost $O(1)$. The complexity of SJA is similar to that of SJ, despite the fact that the space of semijoin-adaptive plans is much larger than the space of semijoin plans (there are $O((m!)2^{m-2})$ semijoin plans assuming we do not consider semijoin plans that are equivalent with respect to our cost model, and there are $O((m!)2^{n(m-2)})$ semijoin-adaptive plans). Moreover, the optimal semijoin-adaptive plan is always at least as good as, and often much better than, the optimal semijoin plan, as we will see in Section 6. So, SJA is preferable to SJ.

The fact that the algorithms presented in this section run in time linear in the number of sources is very important when we deal with a large number of sources as is the case with integrating Internet sources. The running times of SJ and SJA are exponential in the number of conditions. In most realistic scenarios, this is acceptable since the number of conditions (unlike the number of sources) is small. However, if the number of conditions is large then we can use greedy versions of SJ and SJA that work on the conditions in ascending order of *overall selectivity* instead of trying all orderings. For this we define the overall selectivity of a condition c_i as the total number of items at all sources that satisfy c_i. If only communication costs are considered, and they are modeled by a linear function of the number of items being transmitted, the greedy versions preserve the optimality of the SJ and SJA plans. Under our more general cost model, which includes both communication and source processing costs, the greedy versions may produce non-optimal (probably very good) plans. Because they consider only one condition ordering instead of $m!$, the greedy versions run in $O(mn)$ time[4].

[4]The actual run time complexity is $O(mn + m\log m)$, but *logm* is usually much smaller than n.

4 Optimality of Semijoin-Adaptive Plans

In this section, we investigate the optimality of semijoin-adaptive plans within the space of all simple plans. We demonstrate that for queries with less than three conditions, for each simple plan $\mathcal{P}$, there is a semijoin-adaptive plan that performs at least as well as $\mathcal{P}$ on every database. That is, the optimal simple plan falls within the subspace of semijoin-adaptive plans. Later we show that when there are three or more conditions the optimal simple plan may be a non-semijoin-adaptive plan. However, we argue that this non-optimality of semijoin-adaptive plans only occurs in certain "rare" scenarios involving condition correlations.

Before we proceed, we define some notation. Let $c_1, ..., c_m$ $(m \geq 1)$ be an ordering of the conditions of the fusion query. Let $X_1, ..., X_m, X_{11}, ...X_{mn}$ be temporary sets (unary relations). For each $i \in [1..m]$ and $j \in [1..n]$, let s_{ij} be defined by:

$$\begin{array}{ll} s_{1j}: & X_{1j} := sq(c_1, R_j) \\ s_{ij}: & X_{ij} := sq(c_i, R_j) \quad \text{or} \quad X_{ij} := sjq(c_i, R_j, X_{i-1}) \quad \text{if } i > 1 \end{array}$$

and let s'_{ij} be the statement $X_i := X_i \cup X_{ij}$. With this notation we can write a semijoin-adaptive plan as $\mathcal{P} = \mathcal{P}_1; ...\mathcal{P}_m$ where

$$\begin{array}{l} \mathcal{P}_1 = s_{11}; ...; s_{1n}; s'_{11}; ...; s'_{1n} \\ \mathcal{P}_i = s_{i1}; ...; s_{in}; s'_{i1}; ...; s'_{in}\ X_i := X_{i-1} \cap X_i; \quad \text{if } i > 1. \end{array}$$

We also need the auxiliary notion of i-semijoin-adaptive plans. A plan $\mathcal{P}_1; ...; \mathcal{P}_i; \mathcal{P}'$ is *i-semijoin-adaptive* if (i) $\mathcal{P}_1; ...; \mathcal{P}_i$ is semijoin-adaptive for conditions $c_1, ..., c_i$ and (ii) $\mathcal{P}'$ contains no selection or semijoin query in which any condition in $c_1, ..., c_i$ occurs. Note that if m is the number of conditions, an m-semijoin-adaptive plan is semijoin-adaptive.

We first prove an *initialization* lemma that will lead us to our result for fusion queries with fewer than three conditions (Theorem 4.2), and will serve as a basis for an induction in the general case (Theorem 4.5).

Lemma 4.1 Let $\mathcal{P}$ be a simple plan computing a query Q. There exists a 1-semijoin-adaptive plan computing Q and costing never more than $\mathcal{P}$ on any database.

Proof: (sketch) First, suppose that there exists some i such that, for each source j, $sq(c_i, R_j)$ occurs in $\mathcal{P}$. Then, it is easy to transform $\mathcal{P}$ into a 1-semijoin-adaptive plan: add in some preamble to $\mathcal{P}$, $X_{ij} := sq(c_i, R_j)$ for each j, and eliminate all selection or semijoin queries for condition c_i using the X_{ij}'s. The resulting plan will be 1-semijoin-adaptive and will perform at least as well as $\mathcal{P}$ on every database.

Now suppose that no such i exists for $\mathcal{P}$. Then, for each i, there exists $j(i)$ such that $sq(c_i, R_{j(i)})$ does not occur in $\mathcal{P}$. One can then construct a database I as follows: For each i, relation $R_{j(i)}$ has a tuple that satisfies c_i and has a merge attribute value of γ, for some constant γ. Besides these, no other tuples exist in the relations. Then, one can show that Q has a nonempty result on I, but that $\mathcal{P}$ returns an empty result for I, a contradiction since $\mathcal{P}$ was supposed to compute Q. □

Theorem 4.2 Let Q be a query with fewer than three conditions and $\mathcal{P}$ a simple plan computing Q. Then, there exists a semijoin-adaptive plan computing Q and costing never more than $\mathcal{P}$ on any database.

Proof: (sketch) By Lemma 4.1, there exists a 1-semijoin-adaptive plan $\mathcal{P}_1; \mathcal{P}'$ computing Q and costing never more than $\mathcal{P}$ on any database. If Q has only one condition, the result is trivially shown. Suppose it has two conditions, c_1 in $\mathcal{P}_1$ and c_2 in $\mathcal{P}'$. Now, this plan is "almost" semijoin-adaptive. Indeed, $\mathcal{P}_1$ is a prefix of a semijoin-adaptive plan and after $\mathcal{P}_1$, we know the set X_1 of items satisfying c_1. Now, consider each source j in turn. One of the following cases arises:

1. $sq(c_2, R_j)$ occurs in $\mathcal{P}'$. Then one can move this selection early and remove all other queries with c_2 to R_j.

2. $sjq(c_2, R_j, X_1)$ occurs in $\mathcal{P}'$. Similarly, one can move this semijoin query early and remove all other queries with c_2 to R_j.

3. Neither of the above two cases holds. Then, $\mathcal{P}$ can get the proper subset of X_1 satisfying c_2 at Source j only by using several semijoin queries. But each item in X_1 would have to be tried. So, one can not increase the cost by introducing $sjq(c_2, R_j, X_1)$ early and removing all queries with c_2 to R_j.

This allows us to obtain a semijoin-adaptive plan computing Q and costing never more than $\mathcal{P}_1; \mathcal{P}'$ on any database. □

Given that Theorem 4.2 asserts the optimality of semijoin-adaptive plans for upto two conditions, it is natural to wonder if they are optimal for an arbitrary number of conditions. It turns out that it is not the case.

Proposition 4.3 There is a query Q with 3 conditions (and 2 sources), a non-semijoin-adaptive plan $\mathcal{P}$, a database I, and a cost model satisfying the properties of Section 2.4, such that every semijoin-adaptive plan computing Q costs strictly more than $\mathcal{P}$ on I.

Proof: (sketch) Let R_1, R_2 be two sources and c_1, c_2, c_3 be three conditions. We describe a database I and then a non-semijoin-adaptive plan $\mathcal{P}$. First, $I(R_1)$ and $I(R_2)$ have 100 million items each, and c_1 has a selectivity of 0.001. For items from R_1 satisfying c_1, c_2 has selectivity 0.1 and c_3 has selectivity 0.9. For items from R_2 satisfying c_1, c_2 has selectivity 0.9 and c_3 has selectivity 0.1. There are no items that satisfy c_1 in both R_1 and R_2. Finally, c_2 and c_3 have overall selectivity of 0.95. This data distribution suggests the non-semijoin-adaptive plan $\mathcal{P}$: (1) compute the items satisfying c_1 using two selection queries; (2) for those items found in R_1 satisfying c_1, consider c_2, then c_3 (by semijoins); (3) for those items found in R_2 satisfying c_1, consider c_3, then c_2 (again by semijoins). One can show that, on database I, $\mathcal{P}$ strictly outperforms any semijoin-adaptive plan, in a natural cost model that only considers communication costs modeled by a linear function of the number of items being transmitted. □

The key feature of the plan in Proposition 4.3 is that after evaluating c_1 at both R_1 and R_2, the next steps do not simply use X_1, the complete set of items satisfying c_1. Instead, the items satisfying c_1 at R_1 are treated differently from those satisfying c_1 at R_2. The only reason why this pays off is that we have strong "inter-source" dependencies, e.g., the items satisfying c_1 at R_1 have different selectivities for c_2 than the other items satisfying c_1 in general. Given a set of conditions $c_1, ..., c_i$, we call the set of items that satisfy the set of conditions over all the sources the *complete set* (for $c_1, ..., c_i$). A set of items that is not complete is said to be *incomplete*. In our example, the plan uses incomplete sets (e.g., the set of items satisfying c_1 at R_1 only and the set of items satisfying c_1 at R_2 only) as inputs for its semijoins. Our next lemma and theorem show that only plans that use incomplete sets can beat all semijoin-adaptive plans. (The proof of the lemma is presented in the appendix.)

Lemma 4.4 Let $\mathcal{P}_1; ...; \mathcal{P}_i; \mathcal{P}'$ be an i-semijoin-adaptive plan computing some query Q and assume further that $\mathcal{P}'$ uses only the complete set X_i of $\mathcal{P}_i$ and no other sets of items introduced in $\mathcal{P}_1; ...; \mathcal{P}_i$. Then there exists a $(i+1)$-semijoin-adaptive plan $\mathcal{P}_1; ...; \mathcal{P}_{i+1}; \mathcal{P}''$ that computes Q and costs never more than the original plan on any input. Furthermore if $\mathcal{P}'$ uses only complete sets as inputs to its semijoin queries then (i) so does $\mathcal{P}''$ and (ii) $\mathcal{P}''$ uses only the complete set X_{i+1} of $\mathcal{P}_{i+1}$ and no other sets of items introduced in $\mathcal{P}_1; ...; \mathcal{P}_{i+1}$.

Theorem 4.5 Consider a simple plan $\mathcal{P}$ computing Q where all inputs to its semijoin queries are complete sets. Then, there exists a semijoin-adaptive plan computing Q and costing never more than $\mathcal{P}$ on any database.

Proof: (sketch) By Lemma 4.1, we can find a 1-semijoin adaptive plan $\mathcal{P}_1; \mathcal{P}'_1$ that is at least as good as $\mathcal{P}$ on any database. Since $\mathcal{P}$ uses only complete sets as inputs to its semijoin queries so will $\mathcal{P}'_1$. In fact, $\mathcal{P}'_1$ uses only the complete set computed by $\mathcal{P}_1$ and ignores all other sets of items computed by $\mathcal{P}_1$. Using Lemma 4.4, we obtain by induction a semijoin-adaptive plan that is at least as good as $\mathcal{P}$ on all databases. $\Box$

Theorem 4.5 tells us that only plans that use incomplete sets can beat semijoin-adaptive plans. We now argue informally that if conditions are independent, plans that use incomplete sets will do no better than plans that use complete sets. Intuitively, there is no benefit to treating the items that satisfy a condition at one source differently from those that satisfy the same condition at a different source. If conditions are independent, the total amount of work cannot be increased if we simply combine these incomplete sets into a complete one, and adjust our plan accordingly.

Furthermore, even if there are some condition correlations, plans that use complete sets may still be as good as any. For example, suppose we only have *inter-condition* dependencies. For instance, the fact that an item satisfies condition c_1 makes it more likely that it will satisfy c_2, independent of the sources involved. Under this kind of correlation, there is again no advantage to using incomplete sets in the plan. Thus, we still expect plans that use complete sets to be as good as any, and by Theorem 4.5, the best semijoin-adaptive plan would be the optimal simple plan. Of course, with *inter-source* dependencies (e.g., an item satisfying c_1 at R_1 makes it much more likely that the same item will satisfy c_2 at R_2) it does pay off to use incomplete sets, as Proposition 4.3 illustrated.

Our discussion leads us to conjecture that under a reasonable cost model, for each query Q and each probabilistic data distribution with no inter-source dependencies, there exists a semijoin-adaptive plan $\mathcal{P}$ computing Q that performs *on average* at least as good as any other simple plan. A proof of the conjecture would require formally defining the probabilistic data distribution model and average case cost which is beyond the scope of the paper.

It should be emphasized that even when inter-source dependencies exist, one might still want to only consider semijoin-adaptive plans for a number of reasons: (i) information about the correlations is unlikely to be available in contexts like the Internet; (ii) the search space to consider (when not limited to semijoin-adaptive plans) becomes prohibitively large; and (iii) the gains will most often be relatively small even with very distorted data distributions.

5 Postoptimization

In this section, we consider a number of postoptimization techniques that can improve the plans generated by the SJA algorithm. First, we describe two of them that we have implemented in an algorithm named SJA+. Then, we briefly discuss the SJA+ algorithm. The section ends with a description of a set of other postoptimization techniques.

Loading entire sources. Instead of sending a set of queries to a source, the mediator may consider issuing a single query to load the entire source contents and using this result to evaluate all the queries of that source. This can be advantageous in fusion queries involving extremely small source databases or large number of conditions.

To illustrate, consider the two queries on R_3 in $\mathcal{P}_1$ (Steps 3 and 8 in Figure 6(a)). Let the cost of loading the entire contents of R_3 be lower than the cost of issuing the two queries on R_3. Plan

1) $X_{11} := sq(c_1, R_1)$
2) $X_{12} := sq(c_1, R_2)$
3) $X_{13} := sq(c_1, R_3)$
4) $X_1 := X_{11} \cup X_{12}$
5) $X_1 := X_1 \cup X_{13}$
6) $X_{21} := sq(c_2, R_1)$
7) $X_{22} := sjq(c_2, R_2, X_1)$
8) $X_{23} := sq(c_2, R_3)$
9) $X_2 := X_{21} \cup X_{22}$
10) $X_2 := X_2 \cup X_{23}$
11) $X_2 := X_2 \cap X_1$

(a) phase 1: $\mathcal{P}_1$

1) $X_{11} := sq(c_1, R_1)$
2) $X_{12} := sq(c_1, R_2)$
3) $Y := lq(R_3)$
4) $X_{13} := sq(c_1, Y)$
5) $X_1 := X_{11} \cup X_{12}$
6) $X_1 := X_1 \cup X_{13}$
7) $X_{21} := sq(c_2, R_1)$
8) $X_{22} := sjq(c_2, R_2, X_1)$
9) $X_{23} := sq(c_2, Y)$
10) $X_2 := X_{21} \cup X_{22}$
11) $X_2 := X_2 \cup X_{23}$
12) $X_2 := X_2 \cap X_1$

(b) loading sources: $\mathcal{P}_{2a}$

1) $X_{11} := sq(c_1, R_1)$
2) $X_{12} := sq(c_1, R_2)$
3) $X_{13} := sq(c_1, R_3)$
4) $X_1 := X_{11} \cup X_{12}$
5) $X_1 := X_1 \cup X_{13}$
6) $X_{21} := sq(c_2, R_1)$
7) $Z_1 := X_1 - X_{21}$
8) $X_{22} := sjq(c_2, R_2, Z_1)$
9) $X_{23} := sq(c_2, R_3)$
10) $X_2 := X_{21} \cup X_{22}$
11) $X_2 := X_2 \cup X_{23}$
12) $X_2 := X_2 \cap X_1$

(c) using difference: $\mathcal{P}_{2b}$

1) $X_{11} := sq(c_1, R_1)$
2) $X_{12} := sq(c_1, R_2)$
3) $Y := lq(R_3)$
4) $X_{13} := sq(c_1, Y)$
5) $X_1 := X_{11} \cup X_{12}$
6) $X_1 := X_1 \cup X_{13}$
7) $X_{21} := sq(c_2, R_1)$
8) $Z_1 := X_1 - X_{21}$
9) $X_{22} := sjq(c_2, R_2, Z_1)$
10) $X_{23} := sq(c_2, Y)$
11) $X_2 := X_{21} \cup X_{22}$
12) $X_2 := X_2 \cup X_{23}$
13) $X_2 := X_2 \cap X_1$

(d) SJA+ choice: $\mathcal{P}_2$

Figure 6: Postoptimization

$\mathcal{P}_{2a}$ in Figure 6(b) is the result of postoptimizing $\mathcal{P}_1$ by loading R_3 and replacing the two queries of $\mathcal{P}_1$ on R_3 by local computation at the mediator. Note that $lq(R_j)$ is a new operation primitive used to represent the operation of loading the entire relation R_j. Also observe that we use $sq(c_i, Y)$ to stand for the local application of the condition c_i on a set Y of items[5].

Using the difference operation. A significant portion of the cost of semijoin queries to sources is for the transmission of the semijoin sets of items. One way to reduce the size of the semijoin sets is to use the set *difference* operation as part of the local computation at the mediator. This is particularly important if some sources do not support semijoins directly and the semijoin operation has to be emulated.

In Section 1, we gave a simple example of postoptimization using the difference operator. Here we give a second example, now couched in our notation. Consider again plan $\mathcal{P}_1$ of Figure 6(a). In Step 7, $\mathcal{P}_1$ issues a semijoin query. At the end of Step 5, X_1 contains all the items that satisfy c_1. In Step 6, X_{21} collects all the items of relation R_1 that satisfy condition c_2. From X_1 and X_{21}, we can find the set of items that have already satisfied c_1 and c_2. These items need not be sent to the source in Step 7, to ascertain the satisfaction of condition c_2 in source relation R_2. Thus, there is no need to send the entire set X_1 as the semijoin input in Step 7. Instead, we can just send $X_1 - X_{21}$. Figure 6(c) shows the resulting plan.

5.1 SJA+

The SJA+ algorithm incorporates the above two postoptimization techniques as follows. First, it mimics SJA to obtain the best semijoin-adaptive plan for the given query. Then, it uses the *difference* operation to prune the semijoin sets, in all the semijoin queries as described above. Finally, it considers the option of loading entire source contents to further improve the plan. Figure 6(d) shows a plan that may be obtained by SJA+, assuming that the plan of Figure 6(a) is obtained by SJA for the same fusion query.

The time complexity of SJA+ is $O((m!)mn + mn)$. The $(m!)mn$ term is the cost of SJA. The second term mn is for postoptimization, computed as follows. The postoptimization phase in SJA+ considers the semijoin queries of $(m-1)$ conditions to be improved upon, by using the difference operation. For each set of semijoin queries corresponding to a condition, SJA+ spends $O(n)$ time reducing the semijoin sets and modifying the semijoin queries appropriately. Thus, the postoptimization using the difference operation takes $O(mn)$ time. Then, for each of the n sources, SJA+ takes $O(m)$ time to decide on replacing all its queries by an lq operation and local computation at the mediator. The actual modification of the plan also takes $O(m)$ per source, because SJA+ will replace m steps by $1 + 2m$ steps (the first to load the entire source and the rest for local computation). Thus, the total postoptimization cost is $O(mn)$. Note that SJA+ has the same order of complexity as SJA. In particular, the postoptimization phase of SJA+ is very efficient.

We note that the postoptimization phase of SJA+ uses operations that are not allowed in simple plans as defined in Section 2. In this sense, SJA+ yields plans outside the space of simple plans. One could have considered this more general class of plans up front, and systematically searched for optimality within that space. We decided not to follow that approach because of the very large number of plans that must be then considered to find an optimal plan. For instance, the extended SJA algorithm would be exponential in n. Given that n is usually large in the application domains of interest to us, such algorithms are infeasible.

[5] Strictly speaking, Y is not a set of items because it may also include values for non-merge attributes on which the condition has to be applied.

5.2 Other Postoptimization Techniques

Here, we briefly describe a few other techniques that can be employed in postoptimizing the plans generated by our algorithms of Section 3.

Loading relevant portions of sources. This technique is similar to loading the entire sources. In the latter, it may be wasteful to load a large portion of a source that does not contribute to the result of the fusion query because it does not satisfy any of the specified conditions. In order to avoid this waste, one may first gather all the items that satisfy the first (perhaps, the most selective) condition. Next, this set of items is sent to the source with a request to return all the relevant attributes of the items. The relevant attributes are those that appear in the conditions specified in the fusion query. Then, the mediator evaluates the rest of the conditions locally. The same can be done for all the sources. This technique can be adopted at any stage in the semijoin-adaptive plan, where a set of items satisfying a subset of conditions has been computed. The case where this subset of conditions is empty corresponds to the technique of loading entire sources discussed earlier.

Issuing disjunctive queries to sources. This is another technique that is in the spirit of loading relevant portions of sources and processing them locally at the mediator. In this technique, the mediator sends a disjunction of some conditions, in a single query to a source. The source returns all the items that satisfy any of the conditions at the source. The mediator then performs local computations on this result instead of sending any selection or semijoin queries to the source. In order for the mediator to be able to do this, the source needs to return additional information along with the result set of items in response to a disjunctive query. This information may be in the form of tags attached to each item in the result, indicating the list of conditions satisfied by the item. The source may instead return non-merge attributes also in the result, that will aid in the mediator locally applying the conditions, one at a time, on the disjunctive query result. This technique requires that sources support disjunctive queries, and provide appropriate additional information to the mediator.

Using semantic information. We do not mean by semantic information here, fragmentation information that is like in well designed distributed database systems. We mean here semantic information that can be more realistically encountered in the Internet setting such as key dependencies, or the absence of some attributes in some sources in the case of heterogeneous schemas. It is easy to take advantage of such knowledge. In the example of the fusion query on DMV sources, if a source does not record traffic violations, it is not necessary to ask that source the set of drivers with a *dui* violation. This technique leads to the elimination of some of the source queries from the plan and thus can reduce its cost.

Caching at the mediator the sets of items from sources. Suppose that the system maintains at the mediator the set of items from each source in relations $I_1, ..., I_n$. Note that it may be reasonable to cache this kind of information at a mediator since it does not depend on any specific fusion query. It may provide speed-up for instance by allowing us to replace a semijoin with some X at source j by a semijoin with $X \cap I_j$ since only items that the source knows of may participate successfully in the semijoin. The consequent reduction in the size of the semijoin sets in semijoin queries will yield improved plans.

Using bit vectors. In this technique, the first condition is sent to each source. Each source returns a bit vector B_j describing the items that passed the condition at that source. The bit vectors are merged at the mediator into a vector B which is sent to each source, along with the list of m conditions to satisfy. Each source then sends back a $k \times m$ bit matrix, where k is the length of B, describing the satisfaction of the m conditions for each item in B. All the rest of the computation can be carried out locally by the mediator. Essentially, this technique is using the ability to process source queries with disjunctive condition specification, along with cooperation among the sources regarding the mapping of sets of data items to bit vectors. It is to be noted that such a technique is rarely feasible in the context of Internet databases since a majority of the data sources do not support such sophisticated queries and operations. However, we can at least consider the idea of using compact representations to pass sets of items between the mediator and the sources as a postoptimizing technique to improve upon the best plan generated by SJA.

6 Performance Evaluation

In this section we study the performance of our optimization algorithms. We wish to address the following questions:

- Knowing the space of plans each algorithm considers, we obviously expect SJA plans to be superior to SJ plans, and in turn superior to FILTER plans. However, the important question here is: do the plans of SJA far outperform the plans of SJ and FILTER? We ask a similar question regarding the performance gains of SJA+ over SJA: what is the extent to which SJA+ improves upon SJA?

- Correlations of query conditions can affect results in two ways, depending on whether the algorithm is aware of the correlations or not. First, if the algorithm is unaware of correlations (it falsely assumes independence, but correlations exist), it may actually yield a plan that is not even optimal in its target class. In this scenario, could it be that, for instance, SJA gives us a *worse* plan than SJ or FILTER? Second, if SJA takes into consideration the correlation information and yields the best semijoin-adaptive plan, this plan may not be the optimal simple plan (see Section 4). How far from the optimal will the plan found by SJA be?

Obviously it is very difficult to answer these questions in absolute terms since there are so many parameters to consider (e.g., source sizes, number of conditions, correlations). Instead, we have considered some "representative scenarios" to gain some insight into the expected answers. In this section we report on a small fraction (due to space limitations) of the representative scenarios we have considered, and we summarize our empirical answers.

Cost functions. To evaluate the costs of our sample plans we use cost functions that compute the costs of selection queries and semijoin queries. To keep the model simple, we assume uniform cost function expressions across all sources. That is, all selection queries use the same expression to compute their costs, while all semijoin queries use a second expression to compute their costs. For each selection query, we charge $q + r + o * \alpha$, where q is the fixed cost for a query, r is the fixed cost of an answer, α is the number of returned items, and o is a proportionality constant. For each semijoin query, we charge $q + i * \beta + r + o * \alpha$ where q, r, o and α are same as before, β is the number of items sent in the semijoin set and i is a proportionality constant. In particular, we choose $q = r = 1$ and $i = o = 1/100$. (If we are considering communication costs, for example, this could mean that 100 result items or 100 semijoin items can be transmitted for the same cost as the header of a query or a result descriptor.)

We compute the sizes of the results of queries from the information about the sizes of the sources and the selectivities of the conditions at the sources. That is, the size of the result of a selection query $sq(c_i, R_j)$ is the product of the size of the source relation R_j and the selectivity of condition

c_i in R_j. In the case of a semijoin query $sjq(c_i, R_j, X)$, the size of the result is the product of the size of R_j, the selectivity of c_i in R_j and the selectivity of the set of items X. The selectivity of a set of items X is approximated as the size of X divided by the total number of entities across all sources.

To illustrate the cost computations, consider a selection query to a source of size 10,000 with a condition of selectivity 0.05 for that source. This query will cost $1+1+5$ (a cost of 1 for the query, 1 for the result header, and 5 for the answer set of items, because there are $10,000*0.05 = 500$ answer items). Next, consider a semijoin query to the same source, with a semijoin set of 1,000 items. Let there be a total of 100,000 items across all sources. Of the $10,000 * 0.05 = 500$ items at the source that satisfy the condition, we expect $500 * (1,000/100,000) = 5$ items to join with the semijoin set sent as input. Thus, the cost of the semijoin query will be $1+1,000/100+1+5/100(\text{rounded up}) = 13$. If the source does not directly support semijoin queries, we need to send 1,000 selection queries, each costing $1 + 1 = 2$ units, for a total of 2,000 units.

Parameters. The main parameters of our model are the number and sizes of sources and the number and selectivities of the conditions. For the results we present here, we chose the following profile:

1. There are 10 sources and the query under consideration has 3 conditions. These are base values; the numbers are varied in the graphs that follow.
2. Half of the sources have 100 entries while the other half have 1,000,000 entries (this is to model a mixture of source sizes).
3. One of the conditions has a selectivity of 0.01 while all others have a selectivity of 0.33 (this is to model mixed selectivities). Each condition has the same selectivity at all the sources.

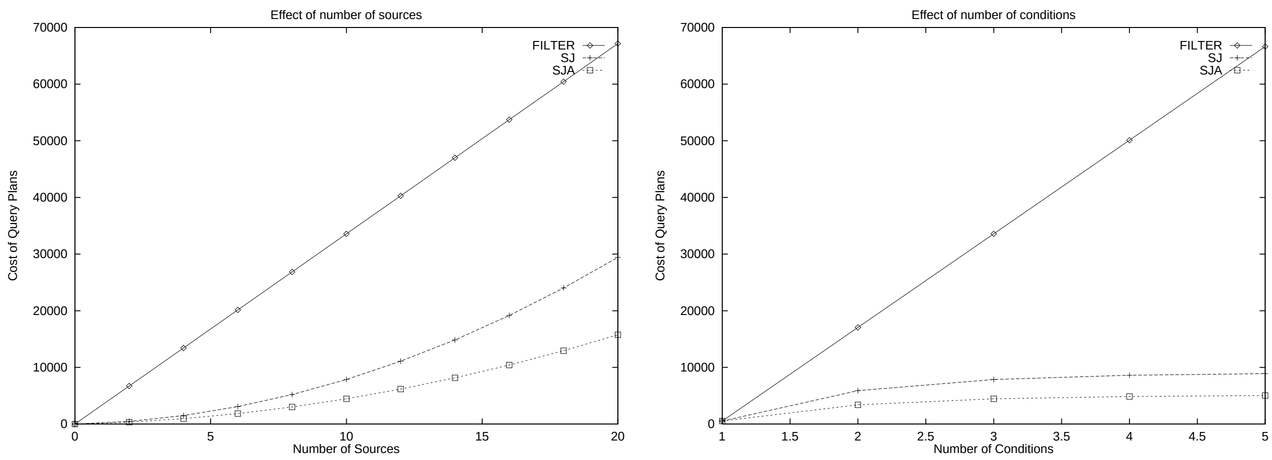
6.1 Comparing the Optimization Algorithms

The behavior of FILTER, SJ, and SJA is shown in Figure 7, as the number of sources and the number of conditions vary. It is clear from these graphs that the FILTER plans are dramatically more expensive than the other plans. In Figure 7(a), it appears that the gap between FILTER and the others narrows as the number of sources increases, but even with 100 sources (not shown), FILTER does substantially worse.

The performance differences between the SJ and SJA plans is not as marked, but it is still significant. For instance, with more than 15 sources, or with 3 conditions or more, the SJA cost is roughly *half* of the SJ plan cost. Although not shown in these graphs, we observed that as source characteristics diverge (in terms of condition selectivities, cost functions), the performance advantages of SJA over SJ are magnified further. For instance, even if 10% of the sources do not support semijoin queries directly (and need expensive emulation at the mediator), the performance of SJ converges to that of FILTER, while the performance of SJA remains relatively unchanged.

We only present the results for varying numbers of sources and conditions, due to space constraints. The results of experiments involving varying condition selectivities and source sizes (not shown here) showed similar trends.

Figure 8(a) illustrates the performance of SJA+ compared to the others, as the number of sources varies. In SJA+ plans, when a mediator loads an entire source (including non-merge attributes), we charge twice the cost per item read. In the figure we see that the SJA+ postoptimization techniques yield significant performance improvements. For instance, with 10 sources, the cost of the SJA+ plan is about 40% of the cost of the SJA plan. More importantly, the cost of SJA+ plans increases linearly with the number of sources, so the advantage of SJA+ becomes



(a) Varying number of sources

(b) Varying number of conditions

Figure 7: Performance of FILTER, SJ and SJA.

significant for large numbers of sources. We also observed in our performance analysis that most of the advantages of SJA+ are due to its use of the difference operation, and not from the option of loading entire source contents.

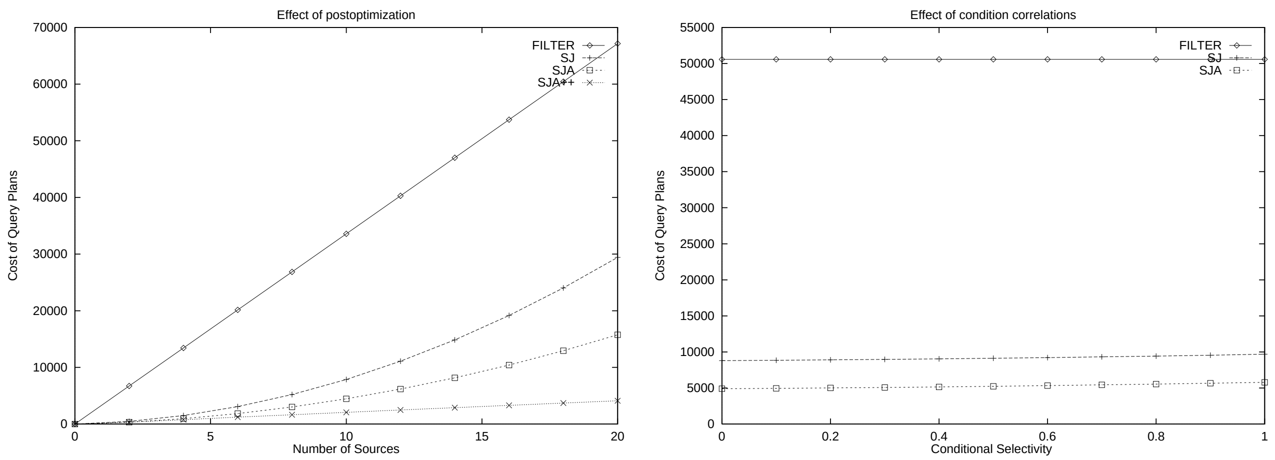
In summary, our results (including many more not presented here) show that the performance differences between the various optimization algorithms can be significant over a wide range of parameter values. Thus, the extra cost of optimizing fusion queries carefully will very likely pay off.

6.2 Correlated Conditions

To illustrate the impact of condition correlations, we set up the following scenario. As before, we have 10 sources and 3 conditions; half of the sources have 100 entries and the other half have 1,000,000 entries. We assume that one condition has a selectivity of 0.01 at all sources and the other two conditions have a selectivity of 0.5 at all sources. We factored in condition correlations by making the second and third condition dependent on the first. That is, we varied the conditional probability of an item satisfying the last two conditions, given that it satisfied the first condition. Varying this probability from 0 to 0.5 models negative correlations and from 0.5 to 1.0 models positive correlations. To magnify the effects of condition correlation we chose similar (both positive or both negative) correlations between the first and second conditions and between the first and third conditions.

Figure 8(b) shows the impact of this conditional selectivity (varying between 0 and 1) on the costs of the plans generated by the FILTER, SJ, and SJA. (We do not consider SJA+ because of the difficulty in estimating its costs under this scenario.) Keep in mind that these optimization algorithms are not aware of the correlations; they generate the same plans as in the previous section.

The correlations do affect the actual costs, but as shown in Figure 8(b), the impact is minimal. The FILTER plans are completely unaffected because condition correlations do not affect selection queries to sources. Plans generated by SJ and SJA are affected, though, because semijoin queries are impacted by correlations. (Positive correlation will increase the size of the result of the semijoin query and may increase the cost, while negative correlation will decrease the size of the semijoin



(a) Postoptimization improvements

(b) Correlated conditions

Figure 8: Postoptimization and condition correlation effects.

query and may decrease the cost.) However, the sensitivity to correlation is small: the cost of both SJ and SJA plans varies by about -5% to +5%. In fact, one has to look very carefully at the graph to see this 5% effect.

Other experiments (whose results are not shown here) confirm that condition correlation rarely has a significant impact on the costs of fusion query plans. This means that even when optimization algorithms use an invalid independence assumption, their performance, relative to other algorithms, and relative to what they could achieve knowing the correlations, does not change much. This implies that it may not be worth gathering condition correlation information and incorporating it into the optimization process (through more accurate cost estimates), given that this task typically increases significantly the cost of the optimization process.

Our final question has to do with whether the cost of the SJA plan shown in Figure 8(b) is far from that of the optimal simple plan. It turns out that in our scenario, the plan generated by SJA with no correlation information, is the same as the plan generated with cost estimates that know about the correlation, and is the same as the optimal simple plan. The latter is not surprising since the correlations in our scenario are inter-condition, not inter-source (see Section 4). So, at least our results confirm the intuition behind the conjecture at the end of section 4. In general, our results show that costs are rather insensitive to correlations, so we expect SJA (and SJA+) to generate excellent fusion query plans, even in the face of condition correlations.

7 Fusion Queries in Existing Optimizers

The expansion of the Internet has led to mediator prototypes that combine information from multiple heterogeneous sources ([3, 6, 14, 20, 22, 34]). Similarly, prototypes for integrating databases have been developed, and recently integration products are being released or announced ([2, 4, 5, 18, 19, 24, 36]).

We note that there is a close connection between mediator-based systems ([39]) and distributed database systems. Given this, many mediator systems have incorporated query processing and optimization techniques of distributed databases. There has been a great deal of published work

$$
\begin{array}{ll}
1) & X_1 := sq(c_1, R_1); Y_1 := sq(c_2, R_1); Z_1 := X_1 \cap Y_1 \\
2) & X_2 := sq(c_1, R_1); Y_2 := sq(c_2, R_2); Z_2 := X_2 \cap Y_2 \\
3) & X_3 := sq(c_1, R_1); Y_3 := sq(c_2, R_3); Z_3 := X_3 \cap Y_3 \\
4) & X_4 := sq(c_1, R_2); Y_4 := sq(c_2, R_1); Z_4 := X_4 \cap Y_4 \\
5) & X_5 := sq(c_1, R_2); Y_5 := sq(c_2, R_2); Z_5 := X_5 \cap Y_5 \\
6) & X_6 := sq(c_1, R_2); Y_6 := sq(c_2, R_3); Z_6 := X_6 \cap Y_6 \\
7) & X_7 := sq(c_1, R_3); Y_7 := sq(c_2, R_1); Z_7 := X_7 \cap Y_7 \\
8) & X_8 := sq(c_1, R_3); Y_8 := sq(c_2, R_2); Z_8 := X_8 \cap Y_8 \\
9) & X_9 := sq(c_1, R_3); Y_9 := sq(c_2, R_3); Z_9 := X_9 \cap Y_9 \\
10) & Z := Z_1 \cup Z_2; Z := Z \cup Z_3; Z := Z \cup Z_4 \\
11) & Z := Z \cup Z_5; Z := Z \cup Z_6; Z := Z \cup Z_7 \\
12) & Z := Z \cup Z_8; Z := Z \cup Z_9
\end{array}
$$

Figure 9: Distributing Join over Union.

on these techniques ([9, 10, 11, 27, 32, 33, 38]). However, most of this work focuses on the efficient evaluation of Select-Project-Join (SPJ) queries. It does not adequately address the special needs of fusion queries.

In this section we study how existing optimizers would handle fusion queries, and we explore opportunities for improvement based on the ideas presented in this paper. The approaches taken by existing optimizers on fusion queries fall into three general categories, so we divide our discussion into three subsections. To illustrate, we continue to use the sample fusion query of Section 1, which finds all the drivers, from three DMV databases, that have "sp" and "dui" violations.

Distribution of the join over the union. The first category contains optimizers that distribute the join operation in a fusion query over the underlying unions. This leads to a plan that is a union of SPJ subqueries, where each SPJ subquery can then be optimized using traditional methods. Such a plan for our sample fusion query is illustrated in Figure 9. In this plan, c_1 is *"$V = sp$"* and c_2 is *"$V = dui$"*. Each line describes a constituent SPJ subplan. The first line, for instance, is a plan for the SPJ subquery that finds the drivers that have both "sp" and "dui" infractions at R_1. The other lines (2 through 9) look for other possible ways to satisfy the query conditions, and the final three lines (10 through 12) union all the possible answers. Notice that there are an exponential number of ways to satisfy the query conditions, and correspondingly there are an exponential number of constituent SPJ subplans.

The plan of Figure 9 happens to be a filter plan. Optimizers in this category may also employ semijoin operations in their query plans. For example, the plan segment that computes Y_3 in the plan of Figure 9 may be replaced by $Y_3 := sjq(c_1, R_1, X_3)$, and this leads to a simple plan that is not a filter plan. Note that plans like the one in Figure 9 are quite inefficient because they repeat many source queries. Common subexpression elimination can be performed to generate more efficient plans. For example, we can eliminate the computation of $Y_4, Y_5, Y_6, Y_7, Y_8, Y_9, X_2, X_3, X_5, X_6, X_8$ and X_9 by reusing common subexpressions Y_1, Y_2, Y_3, X_1, X_4 and X_7. This yields a more efficient filter plan, like the reference filter plans discussed in Section 2. When semijoin operations are used by the optimizers in this category, common subexpression elimination becomes very expensive. This task takes time that is exponential in the number of the constituent SPJ subqueries, which in turn is exponential in the size of the original fusion query.

Examples of systems in our first category are Information Manifold [23], TSIMMIS [29], HERMES [1] and Infomaster [13]. Query processing in these systems is based on resolution ([8, 16]), which leads to the distribution of the join over the union. For example, Information Manifold would process our sample fusion query by generating nine datalog ([37]) subqueries, one for each

$X_1 := sq(c_1, R_1)$	$X_1 := sq(c_1, R_1)$
$X_2 := sq(c_1, R_2)$	$X_2 := sq(c_1, R_2)$
$X_3 := sq(c_1, R_3)$	$X_3 := sq(c_1, R_3)$
$X := X_1 \cup X_2$	$X := X_1 \cup X_2$
$X := X \cup X_3$	$X := X \cup X_3$
$Y_1 := sq(c_2, R_1)$	$Y_1 := sjq(c_2, R_1, X)$
$Y_2 := sq(c_2, R_2)$	$Y_2 := sjq(c_2, R_2, X)$
$Y_3 := sq(c_2, R_3)$	$Y_3 := sjq(c_2, R_3, X)$
$Y := Y_1 \cup Y_2$	$Y := Y_1 \cup Y_2$
$Y := Y \cup Y_3$	$Y := Y \cup Y_3$
$Z := X \cap Y$	$Z := Y$
(a) A Filter Plan	(b) A Semijoin Plan

Figure 10: Handling Unions Uniformly.

of the nine constituent SPJ subqueries in Figure 9. It then unions the results of the nine datalog subqueries to form the answer to the given fusion query.

One obvious way in which systems taking this approach to fusion query processing can incorporate techniques discussed in our paper is to implement a module that checks if a query is a fusion query (by looking for the distinctive pattern of fusion queries) and invokes the algorithm (of Section 3) to generate a semijoin-adaptive plan for the identified fusion query. This leads to a very efficient evaluation of fusion queries without incurring an extremely cumbersome and inefficient optimization process involving common subexpression elimination.

Handling unions uniformly. The second approach to fusion query processing is to separately process the union views, and conceptually generate "temporary" relations. Selection conditions are applied as the temporary relations are computed. Then the temporary relations are joined. Examples of systems using such an approach are DB2 [15] and Tandem's NonStop SQL/MP [35].

Figure 10(a) shows a plan for our sample query generated by this class of optimizers. Here X and Y represent the "temporary relations." The query plans considered by these systems are characterized by the class of filter plans described in Section 2.

This approach of processing the unions uniformly, as described above, does not allow for the use of semijoin operations in the query plans. A slight variation is to combine the steps of union view processing and the join processing. This variation allows for the use of semijoin operations. An example system that uses this variation is Tandem's NonStop SQL/MX [7, 40]. A query plan for our sample query, using this variation, is shown in Figure 10(b).

With this variation, the set of query plans considered includes the class of filter plans and the class of semijoin plans, but not the class of semijoin-adaptive plans. This is because the various sources of a union view are treated homogeneously. That is, if two sources take part in a union view, they both get the same kind of source queries. The technique described in this paper to find the best semijoin-adaptive plan for fusion queries can be employed by systems like Tandem's NonStop SQL/MX without a great deal of effort. The trick is to allow for heterogeneous treatment of the different elements of a union view. That is, one source in the union view may get a selection query while another source in the same union view may get a semijoin query.

Extensible optimizers Recent extensible optimizers ([6, 17, 21, 25]) use flexible, rule-based approaches. The key to efficient fusion query processing in these systems lies in the set of rules defined. Rule-based optimization research has focused on the evaluation of Select-Project-Join queries. We believe that one can write rules, which embody our techniques, to achieve fusion query processing in these optimizers. For example, it is easy to write rules in the Garlic system [20] that help generate efficient filter plans for fusion queries. Combining these with rules for semijoin operations, like the ones given in [20], we can generate semijoin plans for fusion queries. We believe that an extension of these rules (nontrivial, but perhaps not very difficult) may yield semijoin-adaptive plans for fusion queries. Another way to incorporate our techniques into optimizers following the rule-based approach is to have a rule that identifies a fusion query and generates the best semijoin-adaptive plan for it.

As a last remark we note that the techniques described in [1, 12, 26, 41], for estimating the cost of querying the sources, play an important a role in the optimization of fusion queries, just like in the optimization of all queries.

8 Conclusions

Fusion queries are important in environments where data is not well organized and partitioned across autonomous, distributed sites. In this paper we have developed a formal framework for optimizing fusion queries, and we have provided efficient algorithms to produce good query plans over broad scenarios. We have also described enhancements (postoptimizations) that can boost performance significantly, with relatively little additional optimization cost. As discussed in Section 7, our results can be useful for understanding the types of plans current optimizers generate for fusion queries, as well as for improving their performance.

As mentioned earlier, the primary motivation for this work came from the TSIMMIS project which uses a mediator/wrapper approach to integrate data from heterogeneous sources. Fusion queries are central in that context. We are planning to initially incorporate the SJA algorithm into TSIMMIS, followed by some of the postoptimization techniques.

Finally, note that in this paper we focused on minimizing the total work in executing a query. One could also consider minimizing the *response time* of a query in a parallel execution model. It is clear that FILTER plans can easily take advantage of parallelism. However, the plans selected by SJ and SJA can also benefit significantly from parallel execution. SJ and SJA generate query plans that process one condition at a time. For each condition, they issue queries to all the sources. This can be done in parallel, across all the sources. The time to process a particular condition will then be the time taken by the longest source query for that condition.

There are also opportunities for pipelining source queries. For FILTER plans, the pipelining of the queries on each source thread is simple. Pipelining may be more complicated in the case of SJ and SJA, but still possible. For instance, SJA pipelining can be achieved as follows: a mediator process in charge of condition c_i receives streams of items satisfying $c_1, ..., c_{i-1}$ from the various sources; whenever it has obtained a batch of new items, it sends them to all sources that are expecting a semijoin set to handle c_i. Some simple experiments we conducted indicate that SJA, issuing queries in parallel but without pipelining, maintains a large response-time advantage over FILTER, with pipelining on parallel source threads. More experiments and a more comprehensive study of optimization of fusion queries in parallel execution models are needed for a better understanding of all the important issues.

References

[1] S. Adali, S. Candan, Y. Papakonstantinou and V. Subrahmanian. Query Caching and Optimization in Distributed Mediator Systems. In *Proc. ACM SIGMOD Conf.*, pages 137–148, 1996.

[2] R. Ahmed et al. The Pegasus Heterogeneous Multidatabase System. *IEEE Computer*, 24:19–27, 1991.

[3] Y. Arens, C. Chee, C. Hsu and C. Knoblock. Retrieving and Integrating Data from Multiple Information Sources. In *Intl Journal of Intelligent and Cooperative Information Systems*, Vol 2, June 1993.

[4] C. Batini, M. Lenzerini, and S. Navathe. A Comparative Analysis of Methodologies for Database Schema integration. *ACM Computing Surveys*, 18:323–364, 1986.

[5] J. Blakeley. Data Access for the Masses through OLE DB. In *Proc. ACM SIGMOD Conf.*, pages 161–172, 1996.

[6] M. Carey et al. Towards Heterogeneous Multimedia Information Systems: The Garlic Approach. In *Proc. RIDE-DOM Workshop*, pages 124–131, 1995.

[7] P. Celis. The Query Optimizer in TANDEM's New Serverware SQL Product. In *Proc. VLDB Conf.*, 1996.

[8] S. Ceri, G. Gottlobb, and L. Tanca. *Logic Programming and Databases, Surveys in Computer Science.* Springer-Verlag, 1990.

[9] S. Ceri and G. Pelagati. *Database System Concepts.* McGraw Hill, 1997.

[10] A. Chen. Outerjoin Optimization in Multidatabase Systems. In *Proc. Symposium on Databases in Parallel and Distributed Systems*, 1990.

[11] A. Chen and V. Li. Optimizing Star Queries in a Distributed Database System. In *Proc. VLDB Conference*, pages 429-438, 1984.

[12] W. Du, R. Krishnamurthy and M. Shan. Query Optimization in Heterogeneous DBMS. In *Proc. VLDB Conference*, pages 277-291, 1992.

[13] O. Duschka and M. Genesereth. Query Planning in Infomaster. In *Proc. ACM Symposium on Applied Computing*, 1997.

[14] J. Franchitti and R. King. Amalgam: A Tool for Creating Interoperating Persistent, Heterogeneous Components. In *Advanced Database Systems*, pages 313–336, 1993.

[15] P. Gassner, G. Lohman, B. Schiefer and Y. Wang. Query Optimization in the IBM DB2 Family. In *IEEE Data Engineering Bulletin*, 16:4-18, 1993.

[16] M. Genesereth and N. Nillson. *Logical Foundations of Artificial Intelligence.* Morgan Cauffman, 1988.

[17] G. Graefe. The Cascades Framework for Query Optimization. In *Bulletin of the Technical Committee on Data Engineering*, 18:19–29, September 1995.

[18] A. Gupta. *Integration of Information Systems: Bridging Heterogeneous Databases.* IEEE Press, 1989.

[19] P. Gupta and E. Lin. DataJoiner: A Practical Approach to Multidatabase Access. In *Proc. PDIS Conference*, page 264, 1994.

[20] L. Haas, D. Kossman, E. Wimmers, and J. Yang. Optimizing Queries Across Diverse Data Sources. In *Proc. VLDB Conference*, 1997.

[21] L. Haas, J. Freytag, G. Lohman, and H. Pirahesh. Extensible Query Processing in Starburst. In *Proc. ACM SIGMOD Conference*, pages 377–388, 1989.

[22] J. Hammer et al. Information Translation, Mediation, and Mosaic-based Browsing in the TSIMMIS System. In *Proc. ACM SIGMOD Conference*, page 483, May 1995.

[23] A. Levy, A. Rajaraman and J. Ordille. Query Processing in the Information Manifold. In *Proc. VLDB Conference*, 1996.

[24] W. Litwin, L. Mark and N. Roussopoulos. Interoperability of Multiple Autonomous Databases. In *ACM Computing Surveys*, 22:267–293, 1990.

[25] G. Lohman. Grammar-like Functional Rules for Representing Query Optimization Alternatives. In *Proc. ACM SIGMOD Conference*, 1988.

[26] H. Lu, B. Ooi and C. Goh. Multidatabase Query Optimization: Issues and Solutions. In *Proc. RIDE-IMS '93*, pages 137–143, 1993.

[27] T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, 1991.

[28] Y. Papakonstantinou. Query Processing in Heterogeneous Information Sources. Technical report, Stanford University Thesis, 1996.

[29] Y. Papakonstantinou, S. Abiteboul and H. Garcia-Molina. Object Fusion in Mediator Systems. In *Proc. VLDB Conference*, 1996.

[30] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object Exchange Across Heterogeneous information sources. In *Proc. ICDE Conference*, pages 251–260, 1995.

[31] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A query translation scheme for the rapid implementation of wrappers. In *Proc. DOOD Conference*, pages 161–186, 1995.

[32] N. Roussopoulos and H. Kang. A Pipeline N-way Join Algorithm based on the 2-way Semijoin Program. In *IEEE Transactions on Knowledge and Data Engineering*, 3:486-495, December 1991.

[33] A. Silberschatz, H. Korth and S. Sudarshan. *Database System Concepts.* McGraw Hill, 1997.

[34] V. Subrahmanian et al. HERMES: A Heterogeneous Reasoning and Mediator System. http://www.cs.umd.edu/projects/hermes/overview/paper.

[35] S. Sharma. Personal Communication with Sunil Sharma, Tandem Computers Inc. May, 1997.

[36] G. Thomas et al. Heterogeneous Distributed Database Systems for Production Use. *ACM Computing Surveys*, 22:237–266, 1990.

[37] J. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. I.* Computer Science Press, 1988.

[38] C. Wang, A. Chen and S. Shyu. A Parallel Execution Method for Minimizing Distributed Query Response Time. In *IEEE Transactions on Parallel and Distributed Systems*, 3:325-332, May 1992.

[39] G. Wiederhold. Intelligent Integration of Information. In *Proc. ACM SIGMOD Conference*, 1993.

[40] H. Zeller. Personal Communication with Hans Zeller, Tandem Computers Inc. June, 1997.

[41] Q. Zhu and P. Larson. A Query Sampling Method for Estimating Local Cost Parameters in a Multidatabase System. In *Proc. IEEE Data Engineering Conf.*, pages 144–153, 1994.

Appendix: Proof of Lemma 4.4

Statement: Let $\mathcal{P}_1; ...; \mathcal{P}_i; \mathcal{P}'$ be an i-semijoin-adaptive plan computing some query Q and assume further that $\mathcal{P}'$ uses only the complete set X_i of $\mathcal{P}_i$ and no other sets of items introduced in $\mathcal{P}_1; ...; \mathcal{P}_i$. Then there exists a $(i+1)$-semijoin-adaptive plan $\mathcal{P}_1; ...; \mathcal{P}_{i+1}; \mathcal{P}''$ that computes Q and costs never more than the original plan on any input. Furthermore if $\mathcal{P}'$ uses only complete sets as inputs to its semijoin queries then (i) so does $\mathcal{P}''$ and (ii) $\mathcal{P}''$ only uses the complete set X_{i+1} of $\mathcal{P}_{i+1}$ and no other sets of items introduced in $\mathcal{P}_1; ...; \mathcal{P}_{i+1}$.

Proof: We assume without loss of generality that $\mathcal{P}'$ never modifies X_i, that (*) it contains no statement of the form $Y := Z \cup X_i$ (or $Y := X_i \cup Z$); and that (**) it contains no statement of the form $Y := X_i \cap X_i$.

To see (*), observe that we are only interested in items in X_i. So we can (at only local cost), follow any selection statement by an intersection of the result with X_i. Thus before any statement of the form $Y := Z \cup X_i$, Z is a subset of X_i, so we can replace $Z \cup X_i$ by X_i. But we now have to remove statements of the form $Y := X_i$ which are strictly speaking not allowed in simple plans. To do that, it suffices to replace Y by X_i everywhere between an assignment $Y := X_i$ and the next assignment to Y (or the end of the plan). Then we can eliminate the assignment $Y := X_i$. To see (**) $X_i \cap X_i$ can be replaced by X_i and then $Y := X_i$ eliminated.

Let C be the set of conditions in Q and let C' be the conditions that have not been "treated" by $\mathcal{P}_1; ...; \mathcal{P}_i$. We will prove that:

> (***) there exists a (remaining) condition c_{i+1} in C' such that for each j in $[1..n]$ one of the statements of the plan is:
>
> $$Y := sq(c_{i+1}, R_j) \quad \text{or} \quad Y := sjq(c_{i+1}, R_j, X_i)$$
>
> for some variable Y.

Suppose that (***) holds. Let X_{i+1} and for each j, $X_{i+1,j}$ be fresh variables, let $s_{i+1,j}$ be the statement:

$$\begin{array}{lll} X_{i+1,j} := sq(c_{i+1}, R_j); & \text{if } \ Y := sq(c_{i+1}, R_j) & \text{occurs in } \mathcal{P}' \\ X_{i+1,j} := sjq(c_{i+1}, R_j, X_i) & \text{if } \ Y := sjq(c_{i+1}, R_j, X_i) & \text{occurs in } \mathcal{P}' \end{array}$$

and let $s'_{i+1,j}$ be $X_{i+1} := X_{i+1} \cup X_{i+1,j}$. We first introduce at the beginning of $\mathcal{P}'$ the sequence:

$$s_{i+1,1}; ...; s_{i+1,n}; s'_{i+1,1}; ...; s'_{i+1,n}; X_{i+1} := X_i \cap X_{i+1}.$$

Now we can replace every statement of the form $Y := sq(c_{i+1}, R_j)$ or $Y := sjq(c_{i+1}, R_j, Y')$ by $Y := X_{i+1,j}$ or $Y := X_{i+1,j} \cap Y'$, then eliminate statements of the form $Y := X_{i+1,j}$ as done before. It is easy to see that the cost of the new plan is never more than the original plan. (For each j, we have added one costly operation, the computation of $X_{i+1,j}$ but at least removed a statement of identical cost, and possibly others.)

It is now straightforward to see that the resulting plan is a $(i+1)$-semijoin-adaptive plan that satisfies the conditions of the result.

Next, we prove (***) by contradiction. For each c_k in C', let $j(k)$ be such that

$$Y := sq(c_k, R_{j(k)}) \quad \text{and} \quad Y := sjq(c_k, R_{j(k)}, X_i)$$

do not occur for any Y.

Let $Y_1, ..., Y_q$ be the variables of $\mathcal{P}'$ (besides X_i). Let γ be an item. Consider an instance I defined as follows. First, I is such that γ is the single item returned by $\mathcal{P}_1, ...\mathcal{P}_i$. We simply make sure that the tuples that witness that γ is in X_i verify no condition in C'. For each c_k in C', $I(R_{j(k)})$ contains a tuple u_k that has item γ, satisfies c_k and no other condition. I contains no other tuple. We prove by induction that $Y_1, ..., Y_q$ remain empty during the entire computation on input I. Suppose $Y_1, ..., Y_q$ are all empty after s steps of the plan. The next step is one of the following:

- $Y_l := \alpha \cap \beta$. By (**) either α or β is empty, so Y_l remains empty;
- $Y_l := \alpha \cup \beta$. By (*), both α and β are empty, so Y_l remains empty;
- $Y_l := sq(c_k, R_j)$ or $Y_l := sjq(c_k, R_j, X_i)$. Then $j \neq j(k)$ and Y_l remains empty by construction of I.
- $Y_l := sjq(c_k, R_j, Y_{l'})$. Then the result is empty since $Y_{l'}$ is empty.

Thus $Q(I)$ is empty, which contradicts the fact that γ should be in $Q(I)$. Hence (***) holds.

Finally, consider the last sentence of the statement of the lemma. Suppose that $\mathcal{P}'$ uses only complete sets as inputs to its semijoin operators. By construction, $\mathcal{P}''$ also uses only complete sets as inputs to its semijoin operators, so (i) holds. Now consider (ii). By construction, the only temporary relations from $\mathcal{P}_1; ...; \mathcal{P}_{i+1}$ that $\mathcal{P}''$ may use are X_{i+1} and $X_{i+1,j}$ for some j. We show that each occurrence of $X_{i+1,j}$ for each j in $\mathcal{P}''$ can be replaced by X_{i+1} at no extra cost. Othewise, this would indicate that we are using as input to a semijoin a set that would consist of the items in some $X_{i+1,j}$ satisfying some constraint $\mathcal{C}$ but not all items in X_{i+1} satisfying the same constraint, i.e., we would be using an incomplete set as input to a semijoin, a contradiction. Thus, (ii) also holds.