Fusion Queries over Internet Databases *

Ramana Yerneni[1], Yannis Papakonstantinou[2]
Serge Abiteboul[3], Hector Garcia-Molina[4]

[1] yerneni@cs.stanford.edu, Stanford University, USA
[2] yannis@cs.ucsd.edu, University of California, San Diego, USA
[3] serge.abiteboul@inria.fr, INRIA, France
[4] hector@cs.stanford.edu, Stanford University, USA

Abstract. Fusion queries search for information integrated from distributed, autonomous sources over the Internet. 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 good query plans within this large class. We 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, data sources often provide incomplete and 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), Violation (V) and Date of violation (D). Figure 1 shows some sample relations for three DMVs. Now, consider a fusion query that searches for drivers who have both a "driving under the influence" (*dui*) and a "speeding" (*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. 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.

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

```
SELECT  u1.L
FROM    U u1, U u2
WHERE   u1.L = u2.L AND  u1.V = sp AND u2.V = dui
```

* Research partially supported by the Wright Laboratory, Aeronautical Systems Center, Air Force Material Command, USAF, under Grant Number F33615-93-1-1339. This research was done when Papakonstantinou and Abiteboul were at Stanford.

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

Fig. 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 fusion 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, as in the Internet context, 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.

Fusion queries over Internet databases introduce tough performance challenges. Traditional query optimizers do not consider fusion queries in any special way. They treat them simply as queries involving join operations and union views. In addition, many optimizers rely on good fragmentation of data and assume that the number of sources is small. As we have argued earlier, such assumptions are not valid in the Internet context and this often leads conventional techniques to produce poor plans for fusion queries over 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., driver's address) is needed, a "second phase" query would be issued. 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. 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* (Section 2). Simple plans are coordinated by a central site we call the *mediator*. The mediator can ask one or more data 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.[5] 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 sets of values, and sends the entire set to all sources, asking each to select the ones that have $V = sp$. The union of those answer sets would be the final answer. The class of simple plans 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 prove in [24], 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 the space of semijoin-adaptive plans, and still find the best simple plan. Even if the conditions of the query are not independent, 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. Section 3 presents an efficient optimization algorithm to find the best semijoin-adaptive plan for a given fusion query.
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

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

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 $V = dui$. Now, instead of 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 other sources, 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. Postoptimization techniques like this one are discussed in Section 4.

2 Framework

In this section, we provide the framework for fusion query processing. 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* [19] that exports a relation[6]. 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 [18].

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).
- *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.

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:

---

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

```
SELECT  u_1.M
FROM    U u_1, ..., U u_m
WHERE   u_1.M = ... = u_m.M AND  c_1  AND  ...  AND  c_m
```

where each condition c_i, $i = 1, \ldots, m$ involves only one u_i variable and U attributes, and is supported by the wrappers.

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 $X := Y \; op \; Z$, where Y and Z are sets of items, and *op* is either a union ($\cup$) or an intersection ($\cap$). Notice that, in general, mediators perform other operations like joins. However, for fusion queries, union and intersection operations suffice. 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 process 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 to efficiently 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.

Simple plans can be employed in many contexts with a wide range of source capabilities. All that is required is that the sources support selection and semijoin queries. Some sources may not be able to support semijoin queries. In this case, the mediator can emulate a semijoin query as a set of selection queries. The cost of the emulated operation may be higher than if the source supported the semijoin operation, and this will be taken into account in our cost model. If source R_j does not support $sjq(c_i, R_j, Y)$, the mediator can process this semijoin query by sending the source a set of individual selection queries, one for each value in Y. In order for this to work, the source should at least be able to handle selection conditions of the form c_i AND $M = m$, where m is a "passed binding" (from Y). If the source is incapable of supporting even such queries, we can assign an infinite cost to the semijoin query, indicating that it is an unsupported query and hence should not be used in any query plan.

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 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.

2.5 Important Classes of Simple Plans

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

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

Fig. 2. Three simple plans

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. 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.

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 second condition can be evaluated either in a similar fashion or by semijoin queries using X_1 as the 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.

   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., semijoins have to be emulated in an expensive manner) it may not be beneficial to process c_2 at the second source by a semijoin query. 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 a selection query. Thus, the plan can use the best strategy at each source.

3 Finding Optimal Simple Plans

In this section we present three optimization 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.

```
INPUT: Conditions c_1, ..., c_m
       Sources R_1, ..., 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 of operations
              [X_i1 := sq(c_{o_i}, R_1), ..., X_in := sq(c_{o_i}, R_n)]
          append toPlan the operation X_i := X_{i-1} ∩ (∪_{j=1,...,n} X_ij)
          Plan_Cost ← Plan_Cost + selection_queries_cost
        else
          append to Plan the sequence of operations
              [X_i1 := sjq(c_{o_i}, R_1, X_{i-1}), ..., X_in := sjq(c_{o_i}, R_n, X_{i-1})]
          append to Plan the operation 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
```

Fig. 3. The SJ algorithm

We use cost functions $sq\_cost(c_i, R_j)$ and $sjq\_cost(c_i, R_j, X)$ to estimate the costs of the selection query $sq(c_i, R_j)$ and the semijoin query $sjq(c_i, R_j, X)$ respectively. 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. These functions can use whatever information is available at query optimization time, in order to estimate the costs. Techniques like those discussed in [5, 15, 25] can be employed in gathering the relevant statistical information that the cost functions need.

The FILTER algorithm: For a fusion query with m conditions and n sources, the most efficient filter plan is one that issues the mn source queries, pushing each condition to each source, and combining the results of these source queries to compute the answer to the fusion query. FILTER directly outputs such a plan without searching the plan space. Its running time is proportional to the size of the filter plan, which in turn is $O(mn)$, where m is the number of conditions and n is the number of sources.

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

$Optimal\_Plan\_Cost \leftarrow \infty$
for every ordering $[c_{o_1}, \ldots, c_{o_m}]$ of the conditions — *loop A*
  $Plan \leftarrow [X_{11} := sq(c_{o_1}, R_1), \ldots, X_{1n} := sq(c_{o_1}, R_n), X_1 := \cup_{j=1,\ldots,n} X_{1j}]$
  $Plan\_Cost \leftarrow \sum_{j=1,\ldots,n} sq\_cost(c_{o_1}, R_j)$
  for $i = 2, \ldots, m$ — *loop B*
    for $j = 1, \ldots, n$ — *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 $X_{ij} := sq(c_{o_i}, R_j)$
        $Plan\_Cost \leftarrow Plan\_Cost + sq\_cost(c_{o_i}, R_j)$
      else
        append to *Plan* the operation $X_{ij} := sjq(c_{o_i}, R_j, X_{i-1})$
        $Plan\_Cost \leftarrow Plan\_Cost + sjq\_cost(c_{o_i}, R_j, X_{i-1})$
    append to *Plan* the operation $X_i := X_{i-1} \cap (\cup_{j=1,\ldots,n} X_{ij})$
  if $Plan\_Cost < Optimal\_Plan\_Cost$
    $Optimal\_Plan \leftarrow Plan$
    $Optimal\_Plan\_Cost \leftarrow Plan\_Cost$

Fig. 4. The SJA algorithm

The SJ algorithm: SJ (see Figure 3) 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 4) differs from the SJ algorithm in that it makes a separate decision between selection and semijoin query for each condition at each source. In particular, SJA includes the "source loop" of Figure 4 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 shown in [24]. 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 usually small. If the number of conditions is large, one may employ the efficient greedy versions of SJ and SJA that we present in [24]. Those algorithms run in $O(mn)$ time and still find optimal plans under many realistic cost models. However, they may end up with suboptimal, although still very good, plans under the general cost model that we consider here.

4 Postoptimization

In this section, we consider postoptimization techniques that can improve the plans generated by the SJA algorithm. First, we describe two such techniques. Then, we briefly discuss how we efficiently incorporated these techniques in an algorithm named SJA+. In [24], we describe a set of other postoptimization techniques that can further enhance the performance.

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 7 in Figure 5(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 $\mathcal{P}_{2a}$ in Figure 5(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 type used to represent the loading of the entire relation R_j. Also, we use $sq(c_i, Y)$ to stand for the local application of the condition c_i on a set Y of items[7].

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 in the local

[7] 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.

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

(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) $X_1 := X_{11} \cup X_{12} \cup X_{13}$
6) $X_{21} := sq(c_2, R_1)$
7) $X_{22} := sjq(c_2, R_2, X_1)$
8) $X_{23} := sq(c_2, Y)$
9) $X_2 := X_{21} \cup X_{22} \cup X_{23}$
10) $X_2 := X_2 \cap X_1$

(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) $X_1 := X_{11} \cup X_{12} \cup X13$
5) $X_{21} := sq(c_2, R_1)$
6) $Z_1 := X_1 - X_{21}$
7) $X_{22} := sjq(c_2, R_2, Z_1)$
8) $X_{23} := sq(c_2, R_3)$
9) $X_2 := X_{21} \cup X_{22} \cup X_{23}$
10) $X_2 := X_2 \cap X_1$

(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) $X_1 := X_{11} \cup X_{12} \cup X_{13}$
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) $X_{23} := sq(c_2, Y)$
10) $X_2 := X_{21} \cup X_{22} \cup X_{23}$
11) $X_2 := X_2 \cap X_1$

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

Fig. 5. Postoptimization

computations 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 5(a). In Step 6, $\mathcal{P}_1$ issues a semijoin query. At the end of Step 4, X_1 contains all the items that satisfy c_1. In Step 5, 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 R_2 in Step 6, to ascertain the satisfaction of condition c_2. Thus, there is no need to send the entire set X_1 as the semijoin input in Step 6. Instead, we can just send $X_1 - X_{21}$. Figure 5(c) shows the resulting plan.

4.1 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 5(d) shows a plan that may be obtained by SJA+, assuming that the plan of Figure 5(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 $1+m$ 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, a simple direct extension to the SJA algorithm to consider the set difference operations at the mediator would make its time complexity be exponential in n. Given that n is usually large in the application domains of interest to us, such algorithms are infeasible.

5 Fusion Queries in Existing Optimizers

The expansion of the Internet has led to mediator prototypes that combine information from multiple heterogeneous sources ([1, 10, 17, 23]). Similarly, prototypes for integrating databases have been developed, and recently integration products are being released or announced ([2, 9, 13]).

We note that there is a close connection between mediator-based systems 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 on these techniques ([4, 16, 20, 22]). 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.

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. The plans for the constituent SPJ subqueries may involve semijoin operations.

Generating separate subplans for each of the SPJ subqueries can lead to inefficient query plans due to repeated evaluation of common subexpressions. Elimination of common subexpressions can be very cumbersome and expensive, when semijoin operations are used in the subplans. 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 fusion query.

Examples of systems in our first category are Information Manifold [12], TSIMMIS [17], HERMES [23] and Infomaster [6]. Query processing in these systems is based on resolution ([3]), which leads to the distribution of the join over the union.

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 the best semijoin-adaptive plan for the identified fusion query. This leads to a very efficient evaluation of fusion queries without incurring an extremely cumbersome 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 this approach are DB2 [7] and Tandem's NonStop SQL/MP [21]. These systems do not allow for the use of semijoin operations in the query plans. Thus, the plans they consider are characterized by the class of filter plans discussed in Section 2. 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 [21]. 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. Semijoin-adaptive plans can be obtained by allowing 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 ([8, 11, 14]) 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 efficient fusion query processing in these optimizers. For example, it is easy to write rules in the Garlic system [10] that help generate efficient filter plans for fusion queries. Combining these with rules for semijoin operations, like the ones given in [10], 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.

6 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 5, our results can be useful for understanding the types of plans current optimizers generate for fusion queries, as well as for improving their performance.

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. This is a future direction of work we plan to undertake. Another important area of exploration involves moving away from the "two-phase" approach to fusion query processing (as discussed in Section 1). Then we need to consider query plans involving source queries that return other attributes in addition to the merge attributes and this takes us out of the space of simple plans. The techniques we have developed here may still be quite useful in finding very good plans in that more general space of fusion query plans.

References

1. Y. Arens, C. Chee, C. Hsu and C. Knoblock. Retrieving and Integrating Data from Multiple Information Sources. In *Journal of Intelligent and Cooperative Information Systems*, Vol. 2, June 1993.
2. J. Blakeley. Data Access for the Masses through OLE DB. In *Proc. ACM SIGMOD Conf.*, 161–172, 1996.
3. S. Ceri, G. Gottlobb, and L. Tanca. *Logic Programming and Databases, Surveys in Computer Science*. Springer-Verlag, 1990.
4. S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems*. McGraw-Hill, 1984.

5. W. Du, R. Krishnamurthy and M. Shan. Query Optimization in Heterogeneous DBMS. In *Proc. VLDB Conference*, 277-291, 1992.
6. O. Duschka and M. Genesereth. Query Planning in Infomaster. In *Proc. ACM Symposium on Applied Computing*, 1997.
7. 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.
8. G. Graefe. The Cascades Framework for Query Optimization. In *Bulletin of the Technical Committee on Data Engineering*, 18:19–29, September 1995.
9. P. Gupta and E. Lin. DataJoiner: A Practical Approach to Multidatabase Access. In *Proc. PDIS Conference*, 264–264, 1994.
10. L. Haas, D. Kossman, E. Wimmers, and J. Yang. Optimizing Queries across Diverse Data Sources. In *Proc. VLDB Conference*, 1997.
11. L. Haas, J. Freytag, G. Lohman, and H. Pirahesh. Extensible Query Processing in Starburst. In *Proc. ACM SIGMOD Conference*, 377–388, 1989.
12. A. Levy, A. Rajaraman and J. Ordille. Query Processing in the Information Manifold. In *Proc. VLDB Conference*, 1996.
13. W. Litwin, L. Mark and N. Roussopoulos. Interoperability of Multiple Autonomous Databases. In *ACM Computing Surveys*, 22:267–293, 1990.
14. G. Lohman. Grammar-like Functional Rules for Representing Query Optimization Alternatives. In *Proc. ACM SIGMOD Conference*, 1988.
15. H. Lu, B. Ooi and C. Goh. Multidatabase Query Optimization: Issues and Solutions. In *Proc. RIDE-IMS '93*, 137–143, 1993.
16. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, 1991.
17. Y. Papakonstantinou. Query Processing in Heterogeneous Information Sources. Technical report, Stanford University Thesis, 1996.
18. Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object Exchange across Heterogeneous Information Sources. In *Proc. ICDE Conference*, 251–260, 1995.
19. Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A Query Translation Scheme for the Rapid Implementation of Wrappers. In *Proc. DOOD Conference*, 161–186, 1995.
20. 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.
21. S. Sharma and H. Zeller. Personal Communication with Sunil Sharma and Hans Zeller, Tandem Computers Inc. June, 1997.
22. A. Silberschatz, H. Korth and S. Sudarshan. *Database System Concepts.* McGraw-Hill, 1997.
23. V. Subrahmanian et al. HERMES: A Heterogeneous Reasoning and Mediator System. http://www.cs.umd.edu/projects/hermes/overview/paper.
24. R. Yerneni, Y. Papakonstantinou, S. Abiteboul and H. Garcia-Molina. Fusion Queries over Internet Databases (Extended Version). http://www-db.stanford.edu/pub/papers/fqo.ps
25. Q. Zhu and P. Larson. A Query Sampling Method for Estimating Local Cost Parameters in a Multidatabase System. In *Proc. ICDE*, 144–153, 1994.