Capability-Sensitive Query Processing on Internet Sources

Hector Garcia-Molina
hector@cs.stanford.edu

Wilburt Labio
wilburt@cs.stanford.edu

Ramana Yerneni
yerneni@cs.stanford.edu

Abstract

On the Internet, the limited query-processing capabilities of sources make answering even the simplest queries challenging. In this paper, we present a scheme called GenCompact *for generating* capability-sensitive *plans for queries on Internet sources. The query plans generated by* GenCompact *have the following advantages over those generated by existing query-processing systems: (1) the sources are guaranteed to support the query plans; (2) the plans take advantage of the source capabilities; and (3) the plans are more efficient since a larger space of plans is examined.*

1. Introduction

Data sources on the Internet have a wide range of query-processing capabilities. Processing queries on such sources poses interesting challenges, as illustrated by the following examples.

EXAMPLE 1.1 (Bookstore): Consider the Internet bookstore BarnesAndNoble (http://barnesandnoble.com). Suppose we are looking for books written by Sigmund Freud or Carl Jung on the topic of dreams. The BarnesAndNoble query interface (as of 1/1/99) does not allow us to search for two authors at once, so a good plan is to break up the query into two. We can first search for ($author$ = "$Sigmund\ Freud$" $\wedge$ $title\ contains$ "$dreams$"); and then for ($author$ = "$Carl\ Jung$" $\wedge$ $title\ contains$ "$dreams$"). We then take the union of the results of the two queries to obtain the answer to the original query.

Most current query-processing systems would be unable to come up with a good plan for this simple example query. Many systems assume that sources have full relational capabilities, and would try sending the full unsupported query to BarnesAndNoble. Systems that do take into account source capabilities consider limited options. For example, in a system like Garlic [4], query conditions are always processed in conjunctive normal form (CNF), so our condition would be transformed to (($author$ = "$Sigmund\ Freud$" $\vee$ $author$ = "$Carl\ Jung$") $\wedge$ ($title\ contains$ "$dreams$")). Garlic realizes that the first clause cannot be sent to BarnesAndNoble, but that the second one can be. It sends the second clause and applies the first one itself. This plan is valid, but extracts over 2,000 entries from BarnesAndNoble. Our two-query plan, on the other hand, extracts fewer than 20 entries. Thus, if we are concerned about the amount of data retrieved, the Garlic plan is not very good.

Of course, a query-processing system that uses disjunctive normal form (DNF) can come up with our plan. However, there are many examples where CNF instead of DNF is the "right" choice, and there are many other examples where neither CNF nor DNF is a good strategy. The key point is that current systems either ignore source capabilities or only consider limited types of plans, leading to query plans that may be infeasible or inefficient. □

EXAMPLE 1.2 (Car shopping guide): Consider a Web source to which one can pose queries regarding cars for sale (e.g., http://www.autobytel.com). Suppose we are looking for information on midsize or compact sedans. In particular, we want Toyotas under $20,000, and BMWs under $40,000. The query condition in this case is: ($style$ = "$sedan$" $\wedge$ ($size$ = "$compact$" $\vee$ $size$ = "$midsize$") $\wedge$ (($make$ = "$Toyota$" $\wedge$ $price$ <= 20000) $\vee$ ($make$ = "BMW" $\wedge$ $price$ <= 40000))). Suppose the query form for the Web source allows users to specify single values for $style$, $make$ and $price$ along with a list of values for $size$. Our query condition cannot be supported directly by the Web source. However, we can break it up into two conditions: ($style$ = "$sedan$" $\wedge$ $make$ = "$Toyota$" $\wedge$ $price$ <= 20000 $\wedge$ ($size$ = "$compact$" $\vee$ $size$ = "$midsize$")) and the corresponding one for BMWs. We can send the corresponding two queries to the Web source, and union their results to obtain the answer to the original query.

In this example, both DNF and CNF query-processing systems produce less desirable plans. In a DNF system, the user query is transformed into one with four terms and the corresponding four queries are sent to the Web source. Our two-query plan above is preferable, although the same amount of data is transferred in both cases. A CNF system converts the query to one with six clauses, only two of which, ($style$ = "$sedan$") and ($size$ = "$compact$" $\vee$ $size$ = "$midsize$"), can be processed directly by the Web source. Consequently, the CNF system may transfer many more entries from the Web source than necessary. □

As illustrated by the above examples, the task of generating efficient feasible query plans can be difficult, when sources have limited query capabilities. This task may be viewed as a "traditional" query-optimization problem in which different plans for the user query are explored, and the one that performs the best (infeasible plans are deemed the worst) is chosen. However, due to the varied query capabilities of Internet sources, it is difficult to generate plans that are a-priori known to be feasible. Also, the size of the plan space for even moderately complex queries can be very large. Existing optimizers adopt ad hoc strategies, like converting condition expressions into some normal forms, to limit the space of plans considered. Consequently, they miss good plans in many situations. In some cases, they choose infeasible plans when feasible plans exist. In other cases, they choose inefficient feasible plans when much more efficient feasible plans exist.

As the payoff for finding efficient feasible plans in the Internet context is very high, we aim to find good plans by *efficiently* exploring a large space of plans. We achieve this goal as follows:

- We propose a simple description language to capture the varied query capabilities of Internet sources.
- We use this simple description language to determine the feasibility of a query plan very efficiently. Thus, we guarantee that the plans we generate are supported by the sources.
- We present a scheme that efficiently explores large spaces of plans by employing special structures and techniques for compactly representing groups of "related" plans. We also develop pruning rules that help significantly in efficient plan generation.

In presenting our work, we first focus on the processing of selection queries (i.e., queries with only select and project operators). Our examples have illustrated that even the seemingly simple task of processing selection queries can be challenging. Finding good query plans for selection queries is absolutely crucial since they are very common in the Internet context, and they form the building blocks of more complex queries. In [2] we show how the techniques we develop for selection queries can be employed in processing more complex queries.

2. Related work

Very few query-processing systems take into account source capabilities. Conventional systems such as System R [9], DB2 [3] and NonStop SQL [10] assume relational source capabilities without limitations. A few new systems like the Information Manifold [8], TSIMMIS [5], Garlic [4] and DISCO [7] have addressed issues surrounding limited source capabilities.

In the Information Manifold and the TSIMMIS systems, one can only process conjunctive queries. That is, one cannot submit queries with unrestricted condition expressions involving $\wedge$'s and $\vee$'s. In contrast, we allow unrestricted condition expressions.

Garlic allows unrestricted condition expressions. The condition expressions are transformed into CNF and then each clause in the CNF expression is considered for evaluation at the source. If the source cannot evaluate a clause, it is evaluated by Garlic itself. As we illustrated in Section 1, using the CNF strategy can lead Garlic to retrieve too much data for processing. Moreover, if none of the clauses in the CNF expression can be evaluated at the source, Garlic attempts to download the entire source. Such an attempt is not only expensive but also may not be allowed by the source. In general, our approach examines more options than Garlic to discover efficient feasible plans.

In the DISCO system, no restrictions are placed on the condition expressions. However, DISCO does not explore the possibility of splitting the condition expression into parts when looking for feasible plans. Only those options in which the source processes the entire condition expression, or no part of it, are considered. This strategy limits DISCO's ability to generate feasible plans for many queries. For instance, DISCO fails to generate feasible plans for both the example queries of Section 1.

Wrappers are often proposed for dealing with heterogeneous and limited source capabilities (e.g., [5]). However, if wrappers are to provide generic relational capabilities for Internet sources, then they need to implement a scheme like the one we describe in Section 6. That is, when a wrapper receives a query, it must find the best way to execute the query at the underlying source, and this is precisely the problem we are addressing in this paper.

3. Notation

Much of the discussion in this paper focuses on the generation of efficient feasible plans for a given *target query* of the form $\pi_A(\sigma_C(R))$.[1] The condition expression C of the target query is represented by a *condition tree* (CT for short). The leaf nodes of a CT represent Boolean conditions, called *atomic conditions*. The non-leaf nodes of a CT represent the Boolean connectors $\wedge$ or $\vee$. Atomic conditions are denoted c or c_i; *condition expressions*, possibly involving $\wedge$'s or $\vee$'s, are denoted C or C_i.

[1] For simplicity of exposition, we model each Internet source as a relation. Our techniques can be extended in a straightforward manner to other contexts, like when sources provide sets of objects, instead of relations.

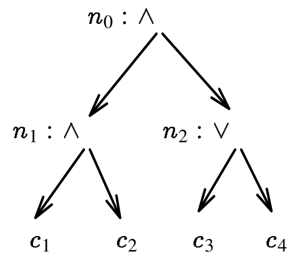

Figure 1. Condition tree.

Figure 1 shows an example of a CT. The condition expression represented by the subtree rooted at some node n of a CT is denoted by $Cond(n)$. For instance, in Figure 1, $Cond(n_1)$ is $(c_1 \wedge c_2)$. Given a condition expression C, we denote the set of attributes appearing in C as $Attr(C)$. $Attr(n)$ is shorthand for $Attr(Cond(n))$. We use $SP(C, A, R)$ as an alternative denotation for $\pi_A(\sigma_C(R))$. In the case of a node n of some CT, $SP(n, A, R)$ is shorthand for $SP(Cond(n), A, R)$.

Unlike conventional relational sources, an Internet source may only support a restricted set of SP queries. Since target queries may not be supported directly by the source, they are submitted to a *mediator* that generates and executes query plans that respect the limitations of the source. A mediator's query plan consists of a set of SP queries to be submitted to the data source and a set of postprocessing operations to be performed by the mediator on the results of the source queries. Typically, the postprocessing operations at a mediator include selection (σ), projection (π), intersection ($\cap$) and union ($\cup$). [2]

EXAMPLE 3.1 (Mediator plans): Consider the target query $SP(((c_1 \wedge c_2) \wedge (c_3 \vee c_4)), A, R)$ whose condition tree is shown in Figure 1. For this target query, the mediator may generate the plan $SP(n_2, A, SP(n_1, A \cup Attr(n_2), R))$. In this plan, the mediator sends the source query $SP(n_1, A \cup Attr(n_2), R)$ and postprocesses its results by applying the selection condition $Cond(n_2)$ and projecting the desired attributes A. Another mediator plan for the same target query can be $SP(n_1, A, R) \cap SP(n_2, A, R)$. This plan has two source queries whose results are intersected by the mediator to arrive at the target-query result. □

4. Describing source capabilities

Internet sources have a wide variety of query-processing limitations. Some of these are:

- *Condition-Attribute Restrictions*: Disallowing condition specification on certain attributes; Requiring that a particular field be filled in.
- *Condition-Expression-Size Restrictions*: Limiting the number of conditions in the condition expression.
- *Condition-Expression-Structure Restrictions*: Allowing only atomic condition expressions; Allowing only conjunctive queries; Restricting expressions based on the structure of a form.

In addition to these, some sources may allow certain attributes to be projected only when appropriate input attributes are specified. For example, a bank may allow the retrieval of some attributes of an account given its account number, but may refuse to give the account balance unless a PIN number is specified in the query condition.

Our source-description language SSDL (for Simple Source-Description Language) is powerful enough to describe the wide range of query capabilities discussed above. Furthermore, as SSDL is based on context-free grammars (CFGs), standard parsing technology can be used to check for the supportability of a source query very efficiently. We now illustrate SSDL before giving its formal definition.

EXAMPLE 4.1 (SSDL): Suppose we have a source $R(make, model, year, color, price)$ that provides information about cars for sale. The query capabilities of R may be described in SSDL as follows.

(1) $\_s \rightarrow \_s_1 \mid \_s_2$
(2) $\_s_1 \rightarrow make = \$m \wedge price < \$p$
(3) $\_s_2 \rightarrow make = \$m \wedge color = \$c$
(4) $attributes :: \_s_1 : \{make, model, year, color\}$
(5) $attributes :: \_s_2 : \{make, model, year\}$

In the above source description, symbols starting with "_" are nonterminals. We use $\$p$ for integer constants; $\$c$ and $\$m$ for string constants. The first three rules are standard CFG rules that describe the condition expressions R can evaluate. For instance, Rule (2) states that R can evaluate conditions like $(make = "BMW" \wedge price < 40000)$ and Rule (3) states that R can evaluate conditions like $(make = "BMW" \wedge color = "red")$.

The last two rules indicate the attributes that can be exported by R. For instance, Rule (4) states that a query matching Rule (2) (i.e., nonterminal $\_s_1$) can fetch the attributes $\{make, model, year, color\}$ (or a subset of them). If the query matches Rule (3) (i.e., nonterminal $\_s_2$), the retrievable attributes are $\{make, model, year\}$. To keep our attribute rules simple, we assume that the *starting nonterminal* $\_s$ can only derive *condition nonterminals* $\_s_j$, and that attribute sets are associated with these condition nonterminals only. Nonterminals that are not directly derived from $\_s$ do not have attribute sets. If the parsing of a query uses condition nonterminal $\_s_j$, then the query may retrieve the attributes that are associated with $\_s_j$. □

[2] The mediator may also perform other operations like duplicate elimination, if necessary.

Formally, the SSDL description of a source R is a triplet $\langle \mathcal{S}, \mathcal{G}, \mathcal{A} \rangle$, where $\mathcal{S}$ is a set of condition nonterminals, $\mathcal{G}$ is the set of CFG rules describing the condition expressions accepted by R, and $\mathcal{A}$ is a set of associations of condition nonterminals to sets of attributes of R. For each condition nonterminal $\_s_i \in \mathcal{S}$, $\mathcal{A}$ must have an association of the form $\_s_i : \{a_{i_1}, \ldots, a_{i_n}\}$, where each a_{i_j} is an attribute of R. The only CFG rule in $\mathcal{G}$ for the starting symbol $\_s$ must be: $\_s \rightarrow \_s_1 \mid \_s_2 \mid \ldots \mid \_s_m$, where $\_s_1$ through $\_s_m$ are the condition nonterminals.

Assuming source R is described in SSDL, we can easily check whether a source query $SP(C, A, R)$ is supported by R. We define a function called $Check$ to verify the supportability of source queries. Given a condition expression and a source, this function returns the set of attributes exported by the source when processing the condition expression. To verify the supportability of the source query $SP(C, A, R)$, a call to $Check(C, R)$ is made. $Check$ uses a parser constructed from the SSDL description of R (using standard tools like YACC [1]) to test if C is valid. If so, $Check$ returns the set of attributes exported by R when processing C. Otherwise, it returns the empty set. If A is a subset of $Check(C, R)$, the source query $SP(C, A, R)$ is deemed supported.

To illustrate the interaction between $Check$ and the generation of feasible plans, consider the target query $SP(C, A, R)$ where $C = ((make = "BMW" \wedge price < 40000) \wedge (color = "red" \vee color = "black"))$ and $A = \{model, year\}$. Figure 1 shows the condition tree for C, with c_1, c_2, c_3 and c_4 representing the four atomic conditions of C. A mediator plan for the target query is feasible if and only if all of its source queries are supported. For instance, consider the plan $SP(n_1, A, R) \cap SP(n_2, A, R)$. A is a subset of $Check(Cond(n_1), R)$, which is $\{make, model, year, color\}$. So $SP(n_1, A, R)$ is a supported query. On the other hand, the second source query $SP(n_2, A, R)$ is not supported because A is not a subset of $Check(Cond(n_1), R)$, the empty set. Hence, this mediator plan is not feasible. Another mediator plan for the same target query is $SP(n_2, A, SP(n_1, A \cup Attr(n_2), R))$. This plan is feasible because its only source query is supported ($A \cup Attr(n_2)$ is a subset of $Check(Cond(n_1), R)$).

5. A modular scheme

In this section, we present a scheme, called *GenModular*, for generating efficient feasible query plans for target queries. This scheme is a naive, exhaustive one. Our goal in presenting it is to show conceptually how one could go about exploring the set of feasible plans for a given target query.

GenModular considers various rewritings of the target-query condition, identifies parts of the condition that can be answered by the source, pieces together the source queries for these parts into query plans for the target query, and chooses the least expensive among these plans. As shown in Figure 2, four modules work together in *GenModular* to achieve these tasks: *rewrite*, *mark*, *generate* and *cost* (hence the "modular" in the name). We describe the rewrite, mark and generate modules in detail. The cost module is not described in detail since its operation is standard. (It selects the best plan from a set of plans, using whatever cost model is applicable.)

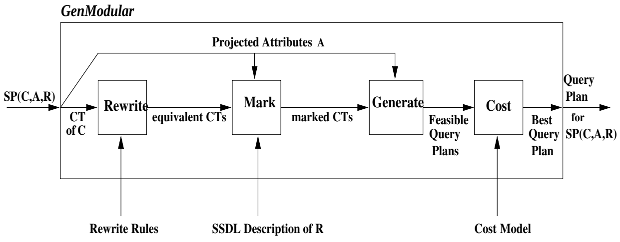

Figure 2. Generating efficient feasible plans.

5.1. Rewrite module

The rewrite module produces a set of equivalent rewritings of the target-query condition. Starting from the CT for the target query, it generates the CTs for the rewritings.

The rewrite module uses a set of rules that are also input to the module. Examples of useful rewrite rules include commutative, associative and distributive transformations of condition expressions. Rules like $C \equiv C \vee C$ and $C \equiv C \wedge C$ (we call them *copy* rules) are also quite useful. In our experimental studies on Internet sources, we observed that using just these rules (commutative, associative, distributive and copy rules) we could generate plans that are far superior to those obtained by contemporary systems (see [2]).

5.2. Mark module

For each CT produced by the rewrite module, the mark module determines the various parts of the CT that can be evaluated at the source. Each node n in the CT has a field $n.export$ that records the set of attributes that can be exported by the source when asked to evaluate $Cond(n)$. By using the $Check$ function, the mark module computes the $export$ fields of all the nodes in the CT. The following example illustrates how CTs are marked in this module.

EXAMPLE 5.1 (Mark): Consider the target query $SP((price < 40000 \wedge color = "red" \wedge make = "BMW"), \{model, year\}, R)$ on R with the SSDL description of Example 4.1. Let t_0 be the CT for the target query.

The rewrite module produces, among others, the CT representing $((make = "BMW" \wedge price < 40000) \wedge (make = "BMW" \wedge color = "red"))$, denoted t_1.

When processing t_1, the mark module invokes $Check(Cond(n_0), R)$ where n_0 is the root node of t_1. R cannot evaluate $Cond(n_0)$, so this call to $Check$ returns the empty set. So, $n_0.export$ is computed to be the empty set.

Notice that the mark module must process *every* CT node even if one of its ancestors represents a condition expression that can be evaluated at R, because we need to explore the possibility of evaluating any part of the CT at R and evaluating the rest of the CT at the mediator.

Continuing with our example, let the first child of n_0 be n_1 (representing $(make = "BMW" \wedge price < 40000)$). $Cond(n_1)$ can be evaluated at R and the exported set of attributes $\{make, model, year, color\}$ is stored in $n_1.export$. Similarly, for the other child of n_0, denoted n_2, the mark module computes $n_2.export$ as $\{make, model, year\}$. All other nodes of t_1 will end up with empty $export$ fields, because none of the condition expressions represented by these nodes can be evaluated at R. For the same reason, the $export$ fields of all the nodes of t_0 are computed to be empty, denoting that no part of t_0 can be evaluated at R. $\square$

5.3. Generate module

Here, we present an algorithm called *EPG* (for Exhaustive Plan Generator) that computes the feasible plans for a given CT. The generate module produces the set of feasible plans by repeatedly invoking *EPG* on each of the CTs passed on by the mark module. The following example illustrates how *EPG* works.

EXAMPLE 5.2 (Generate): Consider the two CTs, t_0 and t_1 of Example 5.1. *EPG* performs a pre-order traversal of each CT. Focusing on t_1, when *EPG* processes n_0, it checks if $n_0.export$ contains the requested attributes of the target query, denoted A. If so, it generates the plan $SP(n_0, A, R)$. We call this type of plan a *pure* plan for n_0 because $Cond(n_0)$ is evaluated in its entirety at the source.

However, $n_0.export$ is empty in our example, so the pure plan for n_0 is not feasible. Regardless of whether this plan is feasible or not, *EPG* examines the children of n_0 to find other plans, because it aims to generate all the feasible plans from the CT. Plans can be generated for n_0 by recursively computing plans for each of its children nodes (i.e., n_1 and n_2) and combining them together. For instance, the following feasible plan is thus constructed for n_0: $SP(n_1, A, R) \cap SP(n_2, A, R)$. We call such a plan *impure* for n_0 because $Cond(n_0)$ is evaluated piecemeal.

There can be many other feasible impure plans for n_0. For instance, source R supports the query $SP(n_1, \{make, model, year, color\}, R)$. Then $Cond(n_2)$ can be locally evaluated at the mediator based on the result of this source query, instead of remotely at the source. This leads to the following feasible plan: $SP(n_2, A, SP(n_1, A \cup Attr(n_2), R))$. In general, for each node like n_0, *EPG* generates plans where a subset of n_0's children are evaluated at the source, and the rest are evaluated locally at the mediator.

EPG generates the above two feasible plans based on t_1. No feasible plans can be generated from t_0 because all the nodes in t_0 have empty $export$ fields indicating that no part of that CT can be evaluated at R. $\square$

We now discuss the formal specification of *EPG* (Algorithm 5.1). Each call to *EPG* is a request to generate a set of plans for a query $SP(n, A, R)$, where n is a node of a CT. *EPG* uses the *Choice* operator to concisely represent the set of plans it generates for n. The children of the $Choice$ operator represent the set of alternative plans for $SP(n, A, R)$. The $Choice$ operator is resolved by the cost module which picks the best among the set of alternative plans.

Algorithm 5.1 *EPG*
Input n, A, R
Output The set of feasible query plans for $SP(n, A, R)$
Method
1. $PLANS \leftarrow \{\}$
2. **if** $(A \subseteq n.export)$ **then**
   3. add $SP(n, A, R)$ to $PLANS$
4. **if** n is an $\wedge$ node **then**
   5. add $\bigcap_{n' \in n.children} EPG(n', A, R)$ to $PLANS$
   6. **for** each $X \subset n.children, X \neq \{\}$
      7. $Local \leftarrow n.children - X$
      8. add $SP(AND(Local), A, \bigcap_{n' \in X} EPG(n', A \cup Attr(AND(Local)), R))\}$ to $PLANS$
9. **else if** n is an $\vee$ node **then**
   10. add $\bigcup_{n' \in n.children} EPG(n', A, R)$ to $PLANS$
11. **if** $((A \cup Attr(n)) \subseteq Check(\mathtt{true}, R))$ **then**
   12. add $SP(n, A, SP(\mathtt{true}, A \cup Attr(n), R))$ to $PLANS$
13. **if** $PLANS = \{\}$ **then return** ϕ
14. **else return** $Choice(PLANS)$

Figure 3. Exhaustive plan generator.

As illustrated in Example 5.2, *EPG* first checks (Line 2) if $SP(n, A, R)$ is supported by R. If so, this pure plan is generated in Line 3. To find the impure plans for an $\wedge$ node n, *EPG* generates plans for each of n's children nodes and combines them to generate plans for n (Line 5). In addition, in Lines 7–8, it generates plans for a subset of n's children and evaluates the rest of the children locally. In Algorithm 5.1, $Local$ denotes the unevaluated children nodes (Line 7); $AND(Local)$ denotes the unevaluated condition (i.e.,

$AND(Local)$ represents the conjunction of each $Cond(m)$ where m is in $Local$). Notice that the recursive *EPG* call in Line 8 must request additional attributes required to evaluate $AND(Local)$.

If n is an $\vee$ node (Lines 9–10), *EPG* again generates a plan for n by combining the plans for all the children of n. Note that there is no opportunity to generate plans that evaluate parts of a disjunction using source queries and evaluate the other parts of the disjunction on the results of these source queries.

Finally, *EPG* explores the possibility of downloading the relevant portion of the source contents and evaluating the condition expression corresponding to n at the mediator. This is done by sending a source query with a trivially true condition and projecting out the requested attributes A and the attributes necessary to evaluate n (Lines 11–12).

If *EPG* finds no plans for n, it returns ϕ to indicate that $SP(n, A, R)$ cannot be evaluated in any way. Any query plan that makes use of ϕ (for instance, when combining the plans from recursive calls to *EPG* in Lines 5, 8 and 10) is automatically eliminated from the set of plans output by the generate module. (This check for empty plans could be done within *EPG*, but we do not show it for simplicity.) If *EPG* finds feasible plans for n, it concisely represents the plans using the $Choice$ operator (Line 14) that indicates the alternative query plans for $SP(n, A, R)$.

6. GenCompact

In the previous section we discussed *GenModular*, a scheme to generate efficient feasible plans for target queries. One drawback of *GenModular* is that it is very inefficient. Here we present a scheme, called *GenCompact*, that can generate the same plans in a much more efficient manner. *GenCompact* improves upon *GenModular* as follows:

- Intelligent Plan Generation: By using a more intelligent plan-generation module that *integrates* the mark, generate and cost modules of *GenModular*, *GenCompact* reduces significantly the number of CTs that need to be processed.

- Pruning Techniques: By using the knowledge of the cost model, *GenCompact* identifies strategies to significantly reduce the plan space explored for each CT, *without* pruning the optimal plan.

We begin our discussion of *GenCompact* by describing the simplified rewrite module in Section 6.1. Section 6.2 presents our cost model upon which the pruning rules of Section 6.3 are based. Finally, in Section 6.4, we present the plan-generation module of *GenCompact*.

6.1. Rewrite module

As in the *GenModular* scheme, *GenCompact* employs a rewrite module to generate a set of CTs equivalent to the CT representing the target-query condition. However, *GenCompact* can work with a lot fewer CTs than *GenModular*. In particular, *GenCompact*'s rewrite module fires fewer rewrite rules, without compromising the optimality of the plans being generated. The commutativity rule is eliminated by rewriting the SSDL description as discussed below. In addition, the associativity and the copy rules are dropped because of *GenCompact*'s improved plan-generation module (described in Section 6.4).

To illustrate how we handle commutativity, consider once again Example 4.1, with the rule:

$$(3)\ \_s_1 \quad \rightarrow \quad make = \$m \wedge color = \$c.$$

This rule specifies that the condition on $make$ must come before the condition on $color$. If the target query condition is ($color$ = "red" $\wedge$ $make$ = "BMW"), it cannot be evaluated at the source. (Of course, for many sources order is not important. However, for this example it is, because that is what the grammar specifies.) In this case, the commutativity rule helped *GenModular* identify the equivalent rewriting ($make$ = "BMW" $\wedge$ $color$ = "red") to generate a feasible plan for the target query. Instead of firing the commutativity rule, one can rewrite R's SSDL description to make R *appear* order insensitive. For instance, given the rule above, we can add the rule:

$$(3')\ \_s_1 \quad \rightarrow \quad color = \$c \wedge make = \$m.$$

When the mediator actually sends a source query, it must "fix" the query to ensure that the correct order is respected. The overhead incurred in fixing source queries is low since the mediator only fixes the source queries of just one plan that is to be executed and not of every possible plan considered. On the other hand, when the source description is not rewritten and the commutativity rule is used, a much larger set of CTs needs to be processed at plan-generation time.

Note that some of the work that used to be performed by the rewrite module of *GenModular* is now going into rewriting the source description, and parsing with a larger set of rules. However, we note that source-description rewriting is undertaken only once when the source is integrated into the system, not every time a target query is processed. Also, by increasing the number of CFG rules in the SSDL description, we only increase the complexity of building the parser, which is done not at run time, but when the source joins the system. When verifying whether a source query is supported, the parser still runs in time linear in the size of the condition expression, irrespective of the number of CFG rules in the source description.

6.2. Cost model

In our cost model, given a plan for the target query that uses a set of source queries SQ, the cost of the plan is:

$$\sum_{sq \in SQ} k_1 + k_2(result\ size\ of\ sq). \tag{1}$$

where k_1 and k_2 are constants that depend on the source referred to by the target query.

Our model approximates to a first degree the communication costs, the source query-processing costs and the mediator costs of postprocessing the source-query results. It is fairly standard to model communication costs with linear functions of the amount of data transferred. Often times, sources impose query-capability limitations based on the set of indexes they have. The cost of processing a feasible query at such a source may be well approximated by a linear function of the size of the query result. In our context, mediators use simple operations like union and intersection to post-process source query results. The cost of such operations may be adequately modeled by a linear function of the size of the data being operated upon.

6.3. Pruning rules

Based on our cost model, we formulate the following pruning rules:

PR_1: *Prune impure plans when pure plan exists.* A pure plan processes the target query entirely at the source (no mediator postprocessing is required). If the pure plan is feasible, no impure plan need be generated, because under our cost model it will never be cheaper than the pure plan. Impure plans use at least as many source queries and transfer at least as much data as the pure plan.

PR_2: *Prune locally sub-optimal plans.* In order to find impure plans for a target query $SP(n, A, R)$, we break it up into many sub-queries, generate plans for the sub-queries and combine them to form the target-query plans. When considering plans to be combined, in our cost model it is safe to prune away all but the cheapest plan for each sub-query.

PR_3: *Prune dominated plans.* Suppose, for example, the target query is $SP((c_1 \wedge c_2 \wedge c_3 \wedge c_4), A, R)$. Let P_1 denote a plan for the sub-query $SP(c_1 \wedge c_2 \wedge c_3, A, R)$ and P_2 denote a plan for the sub-query $SP(c_1 \wedge c_2, A, R)$. If P_2 is at least as expensive as P_1, we say P_1 dominates P_2. In such a case, there is no need to consider P_2 when combining plans to form the target query plan. This is because given any target query plan that uses P_2, we can obtain another plan that is at least as good by replacing P_2 with P_1. In a similar vein, there may be opportunities to prune dominated plans of sub-queries when dealing with a disjunctive target query like $SP((c_1 \vee c_2 \vee c_3 \vee c_4), A, R)$.

These pruning rules are extensively used in *GenCompact*'s plan-generation module and they yield rich dividends (see [2] for performance evaluation of *GenCompact*).

6.4. Plan generation module

The plan-generation module takes each CT produced by the rewrite module and generates a single query plan (without $Choice$ operators) for the CT. After obtaining the best plan for each CT, the overall best plan is chosen.

As indicated earlier, the associativity and copy rules are not used in the rewrite module. To compensate, the plan-generation module has to do more work on each CT it receives from the rewrite module. In particular, for each of its input CTs, it not only considers all the plans that accrue directly from the CT but also explores all the plans that accrue from the CTs that can be obtained by applying the associativity and copy rules on the CT. To facilitate the exploration of all these plans, the plan-generation module starts by converting each CT it receives from the rewrite module into its equivalent *canonical* CT. A CT is in canonical form if the children of every $\wedge$ node are either leaf or $\vee$ nodes and the children of every $\vee$ node are either leaf or $\wedge$ nodes. For instance, the CT representing $(price < 40000 \wedge color = "red" \wedge make = "BMW")$ is canonical because all of the root $\wedge$ node's three children are leaf nodes. On the other hand, the CT representing $(price < 40000 \wedge (color = "red" \wedge make = "BMW"))$ is not canonical since the root $\wedge$ node has two children, one of which is an $\wedge$ node. Converting a CT into an equivalent canonical tree can be done in time linear in the size of the input CT.

6.4.1 Integrated Plan Generator

For each CT received from the rewrite module, the plan-generation module invokes *IPG* (for Integrated Plan Generator) on the corresponding canonical CT, to obtain the best plan for the target query based on that CT.

IPG (see Algorithm 6.1) first checks if the pure plan $SP(n, A, R)$ is feasible. If so, using pruning rule PR_1, *IPG* avoids any further search and returns the pure plan. Otherwise, *IPG* tries to find the cheapest feasible impure plan. One impure plan to consider is to download the relevant portion of the source and evaluate $Cond(n)$ at the mediator. It stores this plan in $plan_{impure}$. If it is not feasible to download the source contents, $plan_{impure}$ will be ϕ, indicating that it is infeasible. Now, if n is a leaf node, there are no other impure plans to be considered. However, if n

Algorithm 6.1 *IPG*
Input n, A , R
Output $plan$ // the best plan for $SP(n, A, R)$
Method
  if $A \subseteq Check(Cond(n), R)$ **then**
    return $SP(n, A, R)$ // the pure plan
  if $A \cup Attr(n) \subseteq Check(\texttt{true}, R)$ **then**
    $plan_{impure} = SP(n, A, SP(\texttt{true}, A \cup Attr(n), R))$
  else // downloading R is not feasible
    $plan_{impure} = \phi$
  if n is a leaf node **then**
    return $plan_{impure}$
  else if n is an $\vee$ node **then**
    Execute code in Figure 5 (Section 6.4.2)
  else if n is an $\wedge$ node **then**
    Execute code in Figure 6 (Section 6.4.3)

Figure 4. Integrated plan generator.

// **Step 1**: Find sub-plans
1. **for** each subset N of $n.children$
  2. $P[N] \leftarrow \phi$
3. **for** each $N \subset n.children$, $N \neq \{\}$
  4. **if** $A \subseteq Check(OR(N), R)$ **then**
    5. $P[N] \leftarrow SP(OR(N), A, R)$
6. **for** each $n' \in n.children$, $P[\{n'\}] = \phi$
  7. $P[\{n'\}] \leftarrow IPG(n', A, R)$
// **Step 2**: Combine the sub-plans
8. Prune all $dominated$ sub-plans from P
9. $E \leftarrow$ subsets N of $n.children$ with $P[N] \neq \phi$
10. **for** each subset SC of E whose union is $n.children$
  11. $plan \leftarrow \bigcup_{N \in SC} P[N]$
  12. **if** $Cost(plan) < Cost(plan_{impure})$ **then**
    13. $plan_{impure} \leftarrow plan$
14. **return** $plan_{impure}$ // best impure plan for n

Figure 5. $\vee$ **node processing of** ***IPG*.**

is not a leaf node, there are opportunities to explore other impure plans. We discuss how these plans are explored and complete the *IPG* algorithm in Sections 6.4.2 and 6.4.3.

Notice that *IPG*, like the *EPG* algorithm of *GenModular* (Section 5.3), traverses the CT in pre-order. Unlike *EPG*, however, *IPG*'s CT traversal can stop without processing all the nodes of the CT, based on pruning rule PR_1.

6.4.2 Processing an $\vee$ node

When *IPG* processes an $\vee$ node and the pure plan is not feasible, it looks for feasible impure plans. There are two steps in finding the best feasible impure plan for an $\vee$ node, n. First, feasible sub-plans that evaluate parts of $Cond(n)$ are identified. Second, the best query plan that combines a subset of the sub-plans is chosen. Figure 5 describes the process more precisely.

Lines 1–7 identify sub-plans for parts of $Cond(n)$ and store them in a sub-plan array P. This array has an entry for each possible subset of n's children. For example, $P[\{n_1, n_2\}]$ stores the best known plan for $Cond(n_1) \vee Cond(n_2)$, where n_1 and n_2 are children of n. In Figure 5, $OR(N)$ denotes the disjunction of each $Cond(n')$ where n' is in N. As usual, ϕ denotes an invalid plan.

Lines 1–5 find pure sub-plans while Lines 6–7 find impure sub-plans. When looking for impure sub-plans, *IPG* does not need to consider subsets of children of n that have more than one element. In addition, if $P[\{n'\}]$ has a feasible pure plan (i.e., $P[\{n'\}]$ is not ϕ at Line 6), *IPG* uses pruning rule PR_1 to avoid searching for impure sub-plans for $\{n'\}$. Otherwise, *IPG* computes the best impure sub-plan for $SP(n', A, R)$ in Line 7. Because of pruning rule PR_2, we can ignore the other sub-plans for n' and keep track of just the one returned by the recursive call to *IPG*.

Note that when $P[\{n'\}]$ is ϕ at Line 6, *IPG* must be called recursively even if there is a valid plan $P[N]$ evaluating more nodes (i.e., $\{n'\} \subset N$). While $P[N]$ evaluates more nodes than the sub-plan that results from the recursive *IPG* call, it may cost more because of increased amount of data transfer. Thus, plans using $P[\{n'\}]$ may be better than plans using $P[N]$ when N is a strict superset of $\{n'\}$.

Once the sub-plans are found, the second step (Lines 8–14) chooses a set of sub-plans with minimum total cost that together evaluate the entire $Cond(n)$. The problem of choosing such a set of sub-plans is the well known "Minimum Cost Set Cover" (MCSC) problem [6]. Since MCSC is known to be NP-complete, to obtain the optimal set of sub-plans, *IPG* examines all possible sets of sub-plans. Assuming there are Q sub-plans, this can be done in $O(2^Q)$ time. In order to keep Q small, *IPG* uses pruning rule PR_3 in Line 8 to discard dominated sub-plans. As shown in [2], the effective use of the pruning rules helps in keeping Q very small for most queries.

Note that, by considering the various sets of sub-plans in Line 9, *IPG* accounts for all the plans that would be considered by *GenModular* from equivalent CTs obtained using the associativity and copy rewrite rules. This is the reason why *GenCompact* is able to drop those rules from its rewrite module without compromising the quality of the target-query plans it finds.

6.4.3 Processing an $\wedge$ node

The plan-generation process to find the best impure plan for an $\wedge$ node is similar to that of an $\vee$ node. That is, the processing is again divided into two steps: (1) find feasible sub-plans for sets of children nodes; and (2) choose the best combination of sub-plans. However, there are significant differences in how the first step is performed as we illustrate in the following example.

EXAMPLE 6.1 ($\wedge$ **node):** Consider the target query $SP((c_1 \wedge c_2 \wedge c_3), A, R)$. Suppose R supports the following three queries: $SP(c_1, A, R)$, $SP(c_2, A \cup Attr(c_3), R)$ and $SP(c_3, A \cup Attr(c_2), R)$.

The pure plan for the target query is not feasible. However, there are three feasible impure plans for the target query based on the source queries supported by R.

$$SP(c_1, A, R) \cap SP(c_2, A, R) \cap SP(c_3, A, R) \quad (2)$$

$$SP(c_1, A, R) \cap SP(c_3, A, SP(c_2, A \cup Attr(c_3), R)) \quad (3)$$

$$SP(c_1, A, R) \cap SP(c_2, A, SP(c_3, A \cup Attr(c_2), R)) \quad (4)$$

We now illustrate how these plans are explored in *IPG*.

The first step in producing the plans for an $\wedge$ node, denoted n, is to find sub-plans that evaluate parts of $Cond(n)$. Just like in the case of an $\vee$ node, *IPG* considers each nonempty subset of n's children, denoted N, and checks if $AND(N)$ can be evaluated at R. In this example case, the only subsets that qualify are: $\{c_1\}$, $\{c_2\}$ and $\{c_3\}$. Unlike in the case of $\vee$ node processing, we have opportunities to process other children of n on the results of source queries with these subsets. For instance, c_3 can be locally evaluated on the result of the source query $SP(c_2, A \cup Attr(c_3), R)$ that supports the subset $\{c_2\}$.

As in the case of the $\vee$ node processing, the sub-plans for the various subsets of the children of n are stored in the sub-plan array P. Notice that, unlike in the $\vee$ node case, for a given subset of children of n, there may exist multiple sub-plans. Using pruning rule PR_2, we keep the best among these and drop others from further consideration. In our example, there are two sub-plans, $SP(c_3, A, SP(c_2, A \cup Attr(c_3), R))$ and $SP(c_2, A, SP(c_3, A \cup Attr(c_2), R))$, for $\{c_2, c_3\}$. If $SP(c_3, A, SP(c_2, A \cup Attr(c_3), R))$ is cheaper than $SP(c_2, A, SP(c_3, A \cup Attr(c_2), R))$, the entries of P are: (1) $P[\{c_1\}] = SP(c_1, A, R)$; (2) $P[\{c_2\}] = SP(c_2, A, R)$; (3) $P[\{c_3\}] = SP(c_3, A, R)$; and (4) $P[\{c_2, c_3\}] = SP(c_3, A, SP(c_2, A \cup Attr(c_3), R))$.

Once all the feasible sub-plans are found, the second step of *IPG* chooses a set of sub-plans that evaluates all the children nodes of n with the minimum cost (again the MCSC problem). Two sets of sub-plans that evaluate all of n's children are found: $\{P[\{c_1\}], P[\{c_2\}], P[\{c_3\}]\}$ and $\{P[\{c_1\}], P[\{c_2, c_3\}]\}$. The first set yields Plan (2) and the second set yields Plan (3). Notice that Plan (4) is not even considered since it is guaranteed to be no more efficient than Plan (3). The costs of both Plans (2) and (3) are evaluated and the cheaper plan is returned. □

// **Step 1**: Find sub-plans
1. **for** each subset N of $n.children$
  2. $P[N] \leftarrow \phi$
3. **for** each $N \subset n.children$, $N \neq \{\}$
  4. **if** $A \subseteq Check(AND(N), R)$ **then**
    5. $P[N] \leftarrow SP(AND(N), A, R)$
    6. $A_N = Check(AND(N), R)$
    7. $N_{add} \leftarrow MaxEval(A_N, n) - N$
    8. **for** each $M \subset N_{add}$, $M \neq \{\}$
      9. $P[N \cup M] \leftarrow min_{cost}(P[N \cup M],$
$SP(\ AND(M),\ A,$
$SP(AND(N), A \cup$
$Attr(AND(M)), R)\ ))$
10. **for** each $n' \in n.children$
  11. **for** each $N' \subset n.children$, $n' \in N'$
    12. **if** there is no N'' such that
$N' \subseteq N''$ and $P[N'']$ is pure **then**
    13. $P[N'] \leftarrow min_{cost}(P[N'],$
$SP(AND(N' - \{n'\}), A,$
$IPG(n', A \cup$
$Attr(AND(N' - \{n'\})), R)))$
// **Step 2**: Combine the sub-plans
14. Prune all $dominated$ sub-plans from P
15. $E \leftarrow$ subsets N of $n.children$ with $P[N] \neq \phi$
16. **for** each subset SC of E whose union is $n.children$
  17. $plan \leftarrow \bigcap_{N \in SC} P[N]$
  18. **if** $Cost(plan) < Cost(plan_{impure})$ **then**
    19. $plan_{impure} \leftarrow plan$
20. **return** $plan_{impure}$ // best impure plan for n

Figure 6. $\wedge$ node processing of *IPG*.

We now discuss in detail the portion of *IPG* that produces the best impure plan for $SP(n, A, R)$ when n is an $\wedge$ node (Figure 6). *IPG* first examines each nonempty subset N of n's children and checks if the query $SP(AND(N), A, R)$ is supported by R. As in Example 6.1, each supported query may produce a set of sub-plans. This set of sub-plans is recorded in the sub-plan array P. Lines 1–2 initialize the sub-plan array and Lines 3–9 compute feasible sub-plans.

For a given supported query $SP(AND(N), A, R)$, denoted q_N, the set of feasible sub-plans based on q_N is computed as follows. The key idea is that the set of attributes A_N that can be exported by R when evaluating $AND(N)$ may be sufficient for the mediator to evaluate other children of n. All such additional children of n are gathered into a set N_{add}. To compute N_{add}, we use a function called $MaxEval$. This function takes the set of attributes A_N and an $\wedge$ node n and returns the set of children of n that can be evaluated using A_N. N_{add} is now simply $MaxEval(A_N, n) - N$. Given N_{add}, we can obtain the various sets of children of n that can be evaluated based on the source query q_N. These sets are computed by Lines 8–9. Notice that when multiple sub-plans exist for a given set of children, Line 9 uses pruning rule PR_2 to prune away all but the cheapest one.

Lines 10–13 generate other feasible sub-plans by invoking *IPG* recursively on each child of n. This is more complicated than in the case when n is an $\vee$ node, because *IPG* looks for opportunities to evaluate sets of children of n, denoted N, based on the plan generated for a particular child of n, denoted n'. Note that in order to evaluate the additional children, the attributes participating in their conditions need to be exported by the plan that evaluates n'. Line 12 uses pruning rules PR_1 and PR_3 to avoid making unnecessary recursive calls to *IPG*. Observe that if $N' = N''$, PR_1 applies and if $N' \subset N''$, PR_3 applies. Line 13 employs pruning rule PR_2 to prune away unnecessary, expensive sub-plans.

Once all the feasible sub-plans are determined, the best combination of the sub-plans that evaluates all the children of n is found by solving the MCSC problem (Lines 15–20). Before solving the MCSC problem, pruning rule PR_3 is employed in Line 14 to eliminate as many sub-plans as possible. To see why there are opportunities for PR_3 in Line 14, note that the application of PR_3 in Line 12 only prunes those impure sub-plans that are dominated by pure sub-plans. In Line 14 we prune impure sub-plans that are dominated by other impure sub-plans. In addition, we prune dominated pure sub-plans.

7. Conclusion

In this paper, we studied the generation of efficient feasible plans for queries over Internet data sources. We presented a simple yet powerful language (SSDL) based on context free grammars to describe the query capabilities of the data sources. We developed an efficient scheme called *GenCompact* that generates good feasible plans.

The quality of the generated plans and the efficiency of the plan-generation process of *GenCompact* were studied through experiments. Due to lack of sufficient space, we have not presented the results of our experiments here. As discussed in the extended version ([2]), our experiments showed that *GenCompact* is not only very efficient but also produces excellent feasible plans for queries over Internet sources with limited capabilities. In [2] we also outline how the techniques developed in this paper can be employed in a modular fashion to extend existing optimizers to better handle queries over limited-capability sources.

GenCompact is a flexible scheme in that it can be easily adapted to situations involving source-capability-description languages and cost models that are different from those presented in this paper. We refer the reader to [2] for a detailed discussion of the flexibility of *GenCompact*.

References

[1] A. Aho, R. Sethi and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1985.

[2] H. Garcia-Molina, W. Labio and R. Yerneni. Capability-Sensitive Query Processing on Internet Sources (Extended Version). http://www-db.stanford.edu/pub/papers/csqpev.ps, 1999.

[3] 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.

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

[5] J. Hammer, S. Nestorov, H. Garcia-Molina, R. Yerneni, M. Breunig and V. Vassalos. Template Based Wrappers in the TSIMMIS System. In *Proc. ACM SIGMOD Conference*, 1997.

[6] D. Hochbaum. Approximation Algorithms for Set Covering and Vertex Cover Problems. In *SIAM Journal of Computing*, 11:555-556, 1982.

[7] O. Kapitskaia, A. Tomasic and P. Valduriez. Dealing with Discrepancies in Wrapper Functionality. *INRIA Research Report RR-3138*, 1997.

[8] A. Levy, A. Rajaraman and J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *Proc. VLDB Conference*, 1996.

[9] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie and T. Price. Access Path Selection in a Relational Database System. In *Proc. ACM SIGMOD Conference*, 1979.

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