Paper Number 156

Capability Sensitive Query Processing on Internet Sources

Hector Garcia-Molina, Wilburt Labio, Ramana Yerneni
Department of Computer Science
Stanford University
contact: yerneni@cs.stanford.edu

Abstract

On the Internet, query processing capabilities of sources may be limited in diverse ways, and this makes answering even the simplest queries challenging. In this paper, we present a scheme called *GenCompact* for generating *capability sensitive* plans for selection queries. The generated query plans may be better than what existing query processing systems produce for three reasons: (1) the sources are guaranteed to support the query plans; (2) the plans take full advantage of the source capabilities; and (3) the plans may be more efficient since a larger space of plans is examined. Even though *GenCompact* considers many plans, it is relatively efficient because it uses effective data structures and pruning rules. We study the optimality of the plans generated as well as the efficiency of the plan generation process.

Keywords: Internet data sources, query processing.

1 Introduction

Data sources over the Internet have a wide range of query processing capabilities. In particular, many sources provide a single-table view, and allow only limited types of selection queries. This introduces interesting query processing challenges, as illustrated by the following examples.

EXAMPLE 1.1 Consider the Internet bookstore Amazon.com, Inc. [1]. Suppose one wants to look for books written by Sigmund Freud or Carl Jung on the topic of dreams. The interface does not allow one to search for two authors at once, so a good plan is to break up the query into two. Thus, we can first search for ($author$ = "$Sigmund\ Freud$" $\wedge$ $title\ contains$ "$dreams$"); and then for ($author$ = "$Carl\ Jung$" $\wedge$ $title\ contains$ "$dreams$"). The results of the two queries can then be unioned 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. Many systems simply assume that sources have full relational capabilities, and would try sending (through a wrapper) the full unsupported query to the Amazon source. Furthermore, systems that do take into account source capabilities, only consider limited options. For example, in a system like Garlic [12], 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 Amazon, but that the second one can be. It thus sends the second clause and applies the first one itself. This plan is valid, but extracts over 2,000 entries from Amazon. The two-query plan, on the other hand, only extracts 9 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 Consider the AutoConnect web site [2] for purchasing cars. One can pose queries to this source regarding cars for sale, using various attributes. Suppose we are looking for information on midsize or compact sedans. In particular, we are interested in Toyotas under \$20,000, and BMWs under \$40,000. The query condition in this case is: $(style = \text{"}sedan\text{"} \wedge (size = \text{"}compact\text{"} \vee size = \text{"}midsize\text{"}) \wedge ((make = \text{"}Toyota\text{"} \wedge price <= 20000) \vee (make = \text{"}BMW\text{"} \wedge price <= 40000)))$. AutoConnect's query form allows users to specify single values for *style*, *make* and *price* along with a list of values for *size*. Hence, our query condition cannot be supported by AutoConnect. However, we can break it up into two conditions: $(style = \text{"}sedan\text{"} \wedge make = \text{"}Toyota\text{"} \wedge price <= 20000 \wedge (size = \text{"}compact\text{"} \vee size = \text{"}midsize\text{"}))$ and the corresponding one for BMWs. We can send these two queries to AutoConnect, 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 AutoConnect. Clearly, the 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, $(style = \text{"}sedan\text{"})$ and $(size = \text{"}compact\text{"} \vee size = \text{"}midsize\text{"})$, can be processed directly by AutoConnect. This leads to many more entries being transferred from AutoConnect than necessary.

Rather than blindly working in DNF or CNF, we believe that a query processing system that deals with Internet sources must carefully consider the space of available options, and select an efficient one that is indeed supported by the source. □

The examples illustrate that even the seemingly simple task of processing selection queries can be complex in an Internet context. In this paper we focus our attention on developing techniques for efficiently generating capability-sensitive plans for such selection queries over Internet data sources. This is because selection queries are very common in the Internet context, and the payoff for identifying good plans for them is very significant, given the slow response time of many Internet sources and the low bandwidth of many connections. In Section 7, we discuss how the selection query techniques can be useful in constructing plans for more complex queries with joins and unions.

The problem we face may be looked at as a "traditional" query optimization problem: We need to explore different plans for our selection query, and choose the one that performs the "best". However, we face some special challenges in this process. The query capabilities of Internet sources vary quite a bit. Some sources process only conjunctive query conditions, while some others process disjunctive conditions in a limited way based on web forms. We have to deal with not only restrictions on the size and structure of condition expressions but also limitations on the sets of attributes returned (see Section 3.3).

The capricious nature of source restrictions makes it very difficult to generate plans that are a-priori known to be feasible. Instead, a huge number of plans must be explored to ensure that we find the one that is feasible and efficient. Existing optimizers adopt ad hoc strategies to limit the space of plans considered, consequently making them unable to find good feasible plans in many situations (see the examples above and Section 2). Because the payoff for finding efficient plans in the Internet context is very high, in this paper, we focus our attention on exploring large spaces of plans for selection queries. We present a scheme that efficiently explores the large spaces of plans by employing special structures and techniques for compactly representing groups of "related" plans, and for determining their feasibility efficiently. We also develop pruning rules specifically targeted for selection queries in the Internet context that help significantly in efficient plan generation.

In Section 3, we define a framework for describing source capabilities and query plans. We present our scheme for generating efficient feasible plans for selection queries in Section 5. Section 4 introduces basic concepts used in our scheme. In Section 6, we study the performance of our scheme through analysis and experiments.

2 Related Work

Very few query processing systems take into account source capabilities. Conventional systems such as System R [26], Ingres [30], DB2 [10] and NonStop SQL [27], as well as others [24,29], assume homogeneous source capabilities. Multi-database systems [23,22,28] address heterogeneity and autonomy, but still assume that "simple" selection queries are supported by all sources.

A few new systems like the Information Manifold [20], TSIMMIS [13,21], Garlic [9,12,32] and DISCO [17,31] have directly addressed source capabilities. In the Information Manifold and TSIMMIS systems, one can only process conjunctive queries. That is, one cannot submit selection queries with arbitrary condition expressions involving $\wedge$'s and $\vee$'s. In contrast, we allow arbitrary condition expressions.

Garlic allows arbitrary selection query 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, this can lead Garlic to retrieve too much data for processing. If none of the clauses in the CNF expression can be evaluated at the source, Garlic attempts to download the entire source. This is not only expensive, but it 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 target query condition expressions, and their framework allows for sources to express disjunctive as well as conjunctive condition processing capabilities. However, DISCO does not explore the possibility of splitting the target query condition into parts when looking for feasible plans. Only those options in which the source is capable of answering the entire condition expression, or none at all, are considered. For instance, both the example queries of Section 1 cannot be executed by the DISCO system.

Other papers like [15,18] also deal with the efficient processing of selection queries. However, the focus of their work is on efficiently evaluating condition expressions involving expensive predicates on a given set of data objects. They do not deal with the issue of finding feasible plans for selection queries in the face of limited source capabilities.

Wrappers are often proposed for dealing with heterogeneous and limited-capability sources (e.g., [14,19]). 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 5. 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 have addressed in this paper.

3 Preliminaries

In this section, we introduce our notation for query plans. The query plans respect the capabilities of the data source as specified using a source description language discussed in Section 3.3.

3.1 Notation

Much of the discussion in this paper focuses on the generation of efficient feasible query plans for a given target selection query q of the form $\pi_A(\sigma_C(R))$. We call R a "source" as well as a "relation".

The selection condition expression C of the target query is represented by a *condition tree* (CT for short). Figure 1 shows an example of a CT. The leaf nodes of a CT represent Boolean conditions, called *atomic conditions*, that do not have $\wedge$'s or $\vee$'s. The non-leaf nodes of a CT represent the Boolean connectors $\wedge$ or $\vee$. (Labels "n_0", "n_1" and "n_2" are used for later reference.)

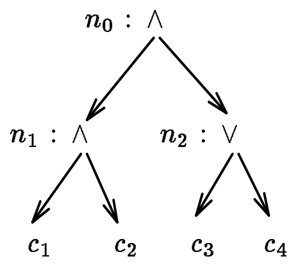

Figure 1: Condition Tree

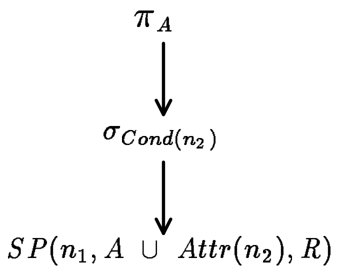

Figure 2: Query Tree

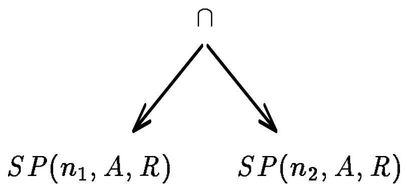

Figure 3: Another Query Tree

Atomic conditions are denoted c or c_i; condition expressions, possibly involving $\wedge$'s or $\vee$'s, are denoted C or C_i. 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 $SP(Cond(n), A, R)$.

3.2 Query Plans

Unlike conventional relational sources, source R may only support a restricted set of SP queries (as described by the source's capability description, see below). Since target queries may not be supported by the source, target queries are submitted to a *mediator* that generates and executes source capability sensitive query plans. A mediator's query plan can be represented by a *query tree*. The leaf nodes represent SP queries, called *source queries*, that are actually sent to R. The non-leaf nodes represent selection (σ), projection (π), intersection ($\cap$) and union ($\cup$) operations that are performed at the mediator to combine the results of the source queries [1]. The non-leaf nodes can also represent a $Choice$ operator as introduced in [11]. A $Choice$ operator is used to concisely represent a set of alternative query plans; each child of the operator represents an alternative query plan. A cost-based module makes the final decision as to which plan to use.

Figures 2 and 3 show examples of query trees representing plans for the target query $SP(((c_1 \wedge c_2) \wedge (c_3 \vee c_4)), A, R)$. Note that n_0, n_1 and n_2 are nodes of the CT shown in Figure 1. Since drawing query trees is not space efficient, we use an "in-line" notation to describe query plans. For instance, $SP(n_2, A, SP(n_1, A \cup Attr(n_2), R))$ is the query plan represented by the query tree in Figure 2 and $SP(n_1, A, R) \cap SP(n_2, A, R)$ is the plan of Figure 3. It may be the case that only one of these query plans is feasible. For instance, if R cannot answer $SP(n_2, A, R)$, then the second plan is not feasible. Also notice that we do not annotate plan operators as to where they are executed: it is always the case that leaf nodes (operating on R directly) are executed at the source, while the rest are executed at the mediator.

3.3 Describing Source Capabilities

Before we discuss how feasible query plans can be generated for a target query, we need a language to describe source capabilities. Using the source capability description, we can check whether a source query is indeed supported by the source. A plan for the target query is feasible if and only if all the source queries it uses are supported by the source.

Internet sources have a wide range of query limitations. Some of these are:

- *Condition Attribute Restrictions*: Disallowing condition specification on certain attributes, e.g., cannot specify a condition on the price attribute of the Barnes and Noble Internet bookstore [3]; Requiring that a particular field be filled in, e.g., must specify stock symbol of a company when querying CheckFree Investment Services stock quote server [7].
- *Limiting the Size of the Condition Expression*: Limiting the number of conditions in the selection condition expression, e.g., one can specify at most three conditions on the advanced search page of the Library of Congress [5].
- *Restrictions on the Structure of the Condition Expression*: Allowing only atomic condition expressions, e.g., most public library web pages [6]; Allowing only conjunctive queries, e.g., the Junglee Internet shopping guide interfaces [4]; Restricting expressions based on the structure of a form, e.g., bookstores like [1] and automobile sales sources like [2].

In addition to these, it is possible that 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.

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

We use a source capability description language based on context free grammars (CFGs) which is powerful enough to describe the selection query capabilities such as the above. At the same time, standard parsing technology can be used to check for the supportability of a query against the capability description very efficiently.

EXAMPLE 3.1 Let us suppose we have a simple source $R(make, model, year, color, price)$. The SP query capabilities of R are described by the following source description.

(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) $ATTRIBUTES :: \_s_1 : \ \{make, model, year, color\}; \ \_s_2 : \ \{make, model, year\}$

Note that symbols starting with "_" are nonterminals. We use $\$p$ for integer constants, and $\$c$ and $\$m$ for string constants. (The parser built from the CFG can perform the necessary type checking.) In the above source description, the first three rules are standard CFG rules that describe the condition expressions R can evaluate. For example, 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 fourth "rule" in our example describes what attributes can be exported by R. It states that a query matching rule (2) (nonterminal $\_s_1$) can fetch the attributes $\{make, model, year, color\}$ (or a subset of them). If the query matches rule (3) (nonterminal $\_s_2$), the retrievable attributes are $\{make, model, year\}$. To keep our attribute rules simple, we assume that in all cases the *starting nonterminal* $\_s$ derives one or more *condition nonterminals* $\_s_j$, and that attribute sets are only associated with these condition nonterminals. Other nonterminals, 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 $\_s_j$. □

We define a function called $Check$ that can be used to test if a selection query is supported by a source. Given a condition expression and a source, this function returns the set of attributes that can be exported by the source when evaluating the condition expression. If the condition expression cannot be evaluated by the source, $Check$ returns the empty set [2]. Thus, a source query $SP(C, A, R)$ is supported by R if $A \subseteq Check(C)$.

To illustrate the interaction between the source description and feasible plan generation, 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. Figures 2 and 3 show two possible query

[2]Under certain unusual circumstances, $Check$ needs to return more than just the attribute set. In particular, if a source can evaluate a condition, without returning any data for matching tuples (it only tells whether matching tuples exist or not), $Check$ needs to return information indicating whether the condition can be evaluated at the source, in addition to identifying the set of exported attributes. We expect such sources to be very rare, and hence work with the simpler $Check$ function in the rest of the paper.

plans for the target query. Notice that the plan represented by Figure 3 is not feasible because $SP(n_2, A, R)$ is not supported by R, i.e., $Check(Cond(n_2))$ returns the empty set. On the other hand, $Check(Cond(n_1))$ returns $\{make, model, year, color\}$. This is a superset of $A \cup Attr(n_2)$. So, $SP(n_1, A \cup Attr(n_2), R)$ is a supported query. Hence, the plan of Figure 2 is feasible.

Other source capability description languages have been proposed in the literature [17,25,33]. For example, [25] uses the relational query description language (RQDL). The CFG based language we discussed above is not as powerful as RQDL in some aspects like describing the join capabilities of sources. But it is more powerful than RQDL in many important aspects of describing selection query capabilities. For instance, CFGs can describe sources that evaluate disjunctive condition expressions while RQDL cannot. Since our focus in this paper is on selection query processing, we use the CFG based language to describe the query capabilities of sources.

4 A Modular Scheme

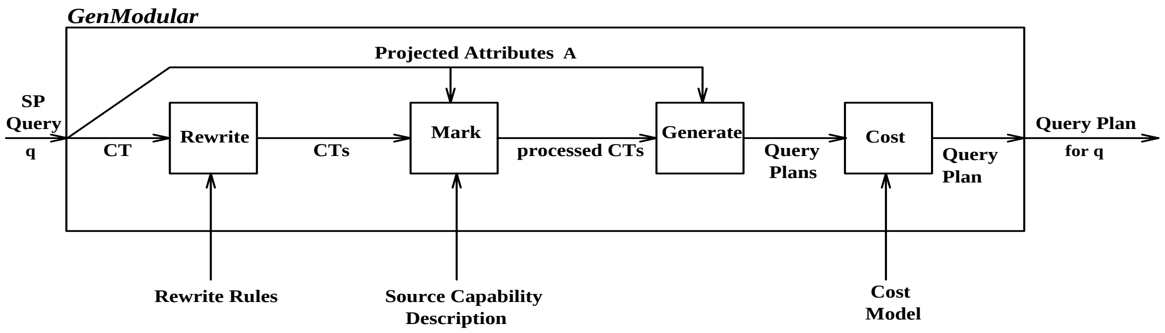

Figure 4: *GenModular* Scheme for Generating Query Plans

In this section, we present a modular scheme, called *GenModular*, for generating efficient feasible query plans for target SP 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. In doing so, we provide a framework that is useful for understanding the much more efficient scheme of Section 5. Thus, keep in mind that for *GenModular* the goal is clarity, not efficiency.

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 4, four modules work together in *GenModular* to achieve these tasks: *rewrite*, *mark*, *generate* and *cost* (hence the "modular" in the name). In this section, 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.)

4.1 Rewrite Module

The rewrite module produces a set of equivalent rewritings of the target query condition. It 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 a set of Internet sources, we observed that using these rules we could generate plans that are far superior to those obtained by contemporary systems (see Section 6).

The target query condition as well as its rewritings are represented by condition trees (CTs). The rewrite module produces a set of CTs that represent condition expressions equivalent to the target query condition. These CTs are passed on to the mark module.

4.2 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. To determine this, the *Check* function (defined in Section 3) is called on the various subtrees of the CT. If a part of the CT can be evaluated at the source, the mark module indicates the set of attributes returned by the source when evaluating that part. The following example illustrates how CTs are marked in this module.

EXAMPLE 4.1 Consider the target query $SP((price < 40000 \wedge color = \text{"}red\text{"} \wedge make = \text{"}BMW\text{"}), \{model, year\}, R)$ and the source description of R given in Example 3.1. Let t_1 be the CT for the target query condition. The rewrite module produces, among others, the CT representing $((make = \text{"}BMW\text{"} \wedge price < 40000) \ \wedge \ (make = \text{"}BMW\text{"} \wedge color = \text{"}red\text{"}))$, denoted t_2.

When processing t_2, the mark module invokes $Check(Cond(n_0))$ where n_0 is the root node of t_2. R cannot evaluate $Cond(n_0)$, so this call to *Check* returns the empty set. Each node n in the CT has a field $n.export$ that records the set of attributes that can be exported by R when asked to evaluate $Cond(n)$. The mark module uses the *Check* function to compute the *export* fields of all the nodes in the CT. In our example, $n_0.export$ 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 that can be evaluated at R. This is 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 the condition $(make = \text{"}BMW\text{"} \wedge price < 40000)$). $Cond(n_1)$ 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_2 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_1 are computed to be empty, denoting that no part of the CT t_1 can be evaluated at R. □

4.3 Generate Module

In this section 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 4.2 Consider the two CTs, t_1 and t_2 of Example 4.1. *EPG* performs a pre-order traversal of each CT. Focusing on t_2, 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* proceeds to examine the children of n_0 to find other plans, because it aims to generate all the feasible plans from the CT. Other 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, this yields the following feasible plan 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_2. No feasible plans can be generated from t_1 because all the nodes in t_1 have empty *export* fields indicating that no part of that CT can be evaluated at R. □

The previous example illustrates how *EPG* generates query plans given a CT. We now discuss the formal algorithm for *EPG* (Algorithm 4.1). 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.

As illustrated in Example 4.2, *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 4.1, *Local* denotes the unevaluated children nodes (Line 7) and $AND(Local)$ 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).

```
Algorithm 4.1  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. $PLANS \leftarrow PLANS \cup \{SP(n, A, R)\}$
    4. if $n$ is an $\wedge$ node then
        5. $PLANS \leftarrow PLANS \cup \{\bigcap_{n' \in n.children} EPG(n', A, R)\}$
        6. for each non-empty proper subset $Remote$ of $n.children$
            7. $Local \leftarrow n.children - Remote$
            8. $PLANS \leftarrow PLANS \cup$
                       $\{SP(AND(Local), A,$
                             $\bigcap_{n' \in Remote} EPG(n', A \cup Attr(AND(Local)), R))\}$
    9. else if $n$ is an $\vee$ node then
        10. $PLANS \leftarrow PLANS \cup \{\bigcup_{n' \in n.children} EPG(n', A, R)\}$
    11. if $((A \cup Attr(n)) \subseteq Check(\mathbf{true}))$ then
        12. $PLANS \leftarrow PLANS \cup \{SP(n, A, SP(\mathbf{true}, A \cup Attr(n), R))\}$
    13. if $PLANS = \{\}$ then return $\phi$
    14. else return $Choice(PLANS)$
```

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 procedure *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 $SP(n, A, R)$.

5 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. In this section, we present a scheme, called *GenCompact*, that can generate the same plans in a much more efficient manner. *GenCompact* improves upon *GenModular* by employing the following:

- Intelligent Plan Generation: By using a more intelligent plan generation module, that integrates the mark, generate and cost modules, *GenCompact* significantly reduces the number of CTs that need to be processed.
- Pruning Techniques: By using the knowledge of the cost model, it identifies strategies to significantly reduce the plan space explored *without* pruning the optimal plan.

The reader should keep in mind that, although *GenCompact* is much more efficient than *GenModular*, it may still be computationally expensive since it must consider many options to find a good plan. The performance characteristics of *GenCompact* are discussed in Section 6.

We begin our discussion of *GenCompact* by describing the simplified rewrite module in Section 5.1. Section 5.2 presents our cost model upon which the pruning rules of Section 5.3 are based.

Finally, in Section 5.4, we present the plan generation module of *GenCompact*.

5.1 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 source description as discussed in Section 5.1.1. In addition, the associativity and the copy rules are dropped because of *GenCompact*'s improved plan generation module (described in Section 5.4). By reducing the set of rewrite rules fired and producing fewer CTs, much of the redundant work performed by *GenModular* is avoided. For instance, in *GenModular*, two CTs that differ by one application of a rewrite rule are processed independently leading to redundant work. This results in *GenCompact* being much more efficient than *GenModular*.

5.1.1 Rewriting Source Capability Description

To illustrate how we handle commutativity, consider once again Example 3.1, with the rule

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

This 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 condition ($make$ = "BMW" $\wedge$ $color$ = "*red*") to generate a feasible plan for the target query. Instead of firing the commutativity rule, one can rewrite R's source description to make R *appear* order insensitive. For instance, given the rule above, we can add the rule

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

Of course, when the mediator actually sends a source query, it must "fix" it so the correct order is respected. The overhead incurred in fixing the source queries is low since the mediator only fixes the source queries of the one query plan that is to be executed and not of every possible query 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 CFG source description, and parsing with a larger set of rules. However, there is an important difference. Source description rewriting is undertaken only once when the source is integrated into the system, not every time a target query is processed. Moreover, by increasing the number of CFG rules, 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 selection condition and not the number of CFG rules in the source description.

5.2 Cost Model

We use a cost model that we believe is well suited for Internet query processing. In this model, given a query plan that uses a set of source queries SQ, the cost of the plan is

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

Constant k_1 represents the overhead in sending the messages over the Internet and the overhead in starting a query at source R (e.g., delay in connecting to the web server). Constant k_2 represents the cost per tuple of computing the answer at source R and the cost per byte of transferring the answer over the network. Note that k_1 and k_2 depend on the source. That is, different sources can have different k_1 and k_2 values in our cost model.

Although mediator costs are not explicitly taken into account in the cost model, minimizing the cost based on Equation (1) will likely lower mediator cost as well. This is because minimizing Equation (1) involves reducing the number of source queries and the amount of data transferred into the mediator. Correspondingly, fewer mediator operations may be required since there are fewer source queries whose results need to be combined. Furthermore, when the results of the source queries are combined, less data needs to be processed.

5.3 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. This is because 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.* This rule applies only to queries with conjunctive conditions (see Section 5.4.3). 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_1 is cheaper than P_2, 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 the cost of any plan P for the target query that uses P_2 can always be lowered by replacing P_2 with P_1 in P.

These three pruning rules are extensively used in the plan generation module of *GenCompact* and they yield rich dividends as evidenced by the performance of *GenCompact* (see Section 6).

5.4 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. As indicated earlier, the associativity and copy rules are not used in the rewrite module. To compensate for this, 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 explore 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.

5.4.1 Integrated Plan Generator

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

```
Algorithm 5.1 IPG
Input $n$, $A$ , $R$
Output plan // the best plan for $SP(n, A, R)$
Method
    if $A \subseteq Check(Cond(n))$ then
      return $SP(n, A, R)$ // the pure plan
    if $A \cup Attr(n) \subseteq Check(\textbf{true})$ then
      $plan_{impure} = SP(n, A, SP(\textbf{true}, A \cup Attr(n), R))$
    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 5.4.2)
    else if $n$ is an $\wedge$ node then
      Execute code in Figure 6 (Section 5.4.3)
```

IPG (see Algorithm 5.1) 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 for the target query. Otherwise, *IPG* tries to find the cheapest feasible impure plan. One impure plan to consider is to download the relevant portions 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 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 5.4.2 and 5.4.3.

Notice that *IPG*, like the *EPG* algorithm of *GenModular* (Section 4.3), 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.

5.4.2 Processing an $\vee$ Node

When *IPG* processes an $\vee$ node and the pure plan is not feasible, it looks for feasible impure plans. The following example illustrates how this is done.

EXAMPLE 5.1 Consider a target query $SP(c_1 \vee c_2 \vee c_3 \vee c_4, A, R)$. Suppose that the following four queries are supported by R: $SP(c_1 \vee c_2, A, R)$, $SP(c_1 \vee c_2 \vee c_3, A, R)$, $SP(c_2 \vee c_3 \vee c_4, A, R)$, $SP(c_1 \vee c_4, B, R)$ (with $B \subset A$).

Since R does not support the condition expression of the target query, there is no feasible pure plan. However, there are a number of feasible impure plans for the target query. The cheapest among these, under our cost model, turns out to be

$$SP(c_1 \vee c_2, A, R) \;\cup\; SP(c_2 \vee c_3 \vee c_4, A, R). \tag{2}$$

We now show how the various query plans are explored and how Plan (2) is chosen.

Let the canonical CT representing the target query condition be t_0. Having determined that the pure plan for the target query is infeasible, *IPG* finds the feasible sub-plans for parts of the disjunctive condition. More precisely, when an $\vee$ node n is processed, *IPG* checks if the disjunction of each non-empty subset of n's children can be evaluated at R. When the root node of t_0 is processed, *IPG* determines that the only four subsets that can be evaluated by R are: $\{c_1, c_2\}$, $\{c_1, c_2, c_3\}$, $\{c_2, c_3, c_4\}$ and $\{c_1, c_4\}$. The corresponding four sub-plans are: $SP((c_1 \vee c_2), A, R)$, $SP((c_1 \vee c_2 \vee c_3), A, R)$, $SP((c_2 \vee c_3 \vee c_4), A, R)$, and $SP((c_1 \vee c_4), B, R)$. The fourth sub-plan is not useful because it does not project enough attributes.

Once the three useful sub-plans are identified, *IPG* constructs query plans for the root node of t_0 by combining the sub-plans using a mediator union (since an $\vee$ node is being processed). Of course, each combination of the sub-plans must "cover" all the children of the root node. Only three combinations of sub-plans qualify:

1. $SP((c_1 \vee c_2), A, R) \cup SP((c_2 \vee c_3 \vee c_4), A, R)$
2. $SP((c_1 \vee c_2 \vee c_3), A, R) \cup SP((c_2 \vee c_3 \vee c_4), A, R)$
3. $SP((c_1 \vee c_2), A, R) \cup SP((c_1 \vee c_2 \vee c_3), A, R) \cup SP((c_2 \vee c_3 \vee c_4), A, R)$.

Notice that the first plan, identical to Plan (2), is clearly the best plan. The second plan is more expensive than Plan (2) since more data is transferred. The third plan is even worse since more data is transferred and more source queries are used. Thus, Plan (2) is chosen. □

As illustrated in the example, when the pure plan is not feasible, there are two steps in finding the best feasible impure plan for an $\vee$ node, n. First, feasible sub-plans that evaluate "parts" of

```
// **Step 1**: Find sub-plans
1. for each non-empty subset $N$ of $n.children$
   2. $P[N] \leftarrow \phi$
3. for each non-empty subset $N$ of $n.children$
   4. if $A \subseteq Check(OR(N))$ then
      5. $P[N] \leftarrow SP(OR(N), A, R)$ // the pure plan for $OR(N)$
6. for each $child$ $n'$ of $n$ such that $P[\{n'\}] = \phi$
   7. $P[\{n'\}] \leftarrow IPG(n', A, R)$ // impure plan for $OR(\{n'\})$

// **Step 2**: Solve the MCSC problem and combine the sub-plans
8. $E \leftarrow$ all subsets $N$ of $n.children$ such that $P[N] \neq \phi$
9. for each subset $SC$ of $E$ that evaluates all of $n$'s children
   10. $plan \leftarrow \bigcup_{N \in SC} P[N]$
   11. if $Cost(plan) < Cost(plan_{impure})$ then
      12. $plan_{impure} \leftarrow plan$
13. Return $plan_{impure}$ // impure plan for $n$
```

Figure 5: $\vee$ Node Processing of *IPG*

$Cond(n)$ are identified. Second, the best query plan that combines a subset of the sub-plans is chosen. Figure 5 describes the process more precisely.

Given a query $SP(n, A, R)$, Lines 1–5 identify the sub-plans for parts of $Cond(n)$ that project at least A 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.

Other sub-plans are obtained, if necessary, by recursively calling *IPG* on each child n' of n (Lines 6–7). This is looking for sub-plans that are not generated by Lines 1–5 in the case of singleton N. If by Line 6 $P[\{n'\}]$ has a valid plan (not ϕ), a pure plan has been found for n', and using pruning rule pr_1, we can avoid searching for impure plans for n'. Otherwise, *IPG* computes the best impure 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 invalid at Line 6, *IPG* still needs to be called recursively even when there is a valid plan $P[N]$ that evaluates 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 8–13) chooses a set of sub-plans with minimum total cost that together evaluate the entire $Cond(n)$. This set of sub-plans are combined using a mediator union (Line 10). The problem of choosing such a set of sub-plans is the well known "Minimum Cost Set Cover" (MCSC) problem [16]. 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. Even though this is exponential in Q, as we discuss in Section 6, for many queries Q turns out to be small.

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.

5.4.3 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 5.2 Consider the target query $SP((c_1 \wedge c_2 \wedge c_3), A, R)$. Suppose that R supports the following three queries: $SP(c_1, A, R)$, $SP(c_2, A \cup \mathit{Attr}(c_3), R)$, $SP(c_3, A \cup \mathit{Attr}(c_2), R)$.

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.

$$SP(c_1, A, R) \cap SP(c_2, A, R) \cap SP(c_3, A, R) \tag{3}$$

$$SP(c_1, A, R) \cap SP(c_3, A, SP(c_2, A \cup \mathit{Attr}(c_3), R)) \tag{4}$$

$$SP(c_1, A, R) \cap SP(c_2, A, SP(c_3, A \cup \mathit{Attr}(c_2), R)) \tag{5}$$

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 $\mathit{Cond}(n)$. Just like in the case of an $\vee$ node, *IPG* considers each non-empty 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 compute feasible sub-plans for other subsets of n. For instance, c_3 can be locally evaluated on the result of the source query $SP(c_2, A \cup \mathit{Attr}(c_3), R)$ that supports the children set $\{c_2\}$. This sub-plan is used in Plan (4).

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, assuming sub-plan $SP(c_3, A, SP(c_2, A \cup \mathit{Attr}(c_3), R))$ is cheaper than sub-plan $SP(c_2, A, SP(c_3, A \cup \mathit{Attr}(c_2), R))$, 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) $P[\{c_2, c_3\}] = SP(c_3, A, SP(c_2, A \cup \mathit{Attr}(c_3), R))$. Given the supported queries we assumed, no other feasible sub-plans exist.

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 (3) and the second set yields Plan (4). Notice that Plan (5) is not even considered since it is guaranteed to be no more efficient than Plan (4). The costs of both Plans (3) and (4) are evaluated and the cheaper plan is returned. □

```
// Step 1: Find sub-plans
1. for each non-empty subset $N$ of $n.children$
   2. $P[N] \leftarrow \phi$
3. for each non-empty subset $N$ of $n.children$
   4. if $A \subseteq Check(AND(N))$ then
      5. $P[N] \leftarrow SP(AND(N), A, R)$ // the pure plan evaluating $AND(N)$
      6. $A_N = Check(AND(N))$
      7. $N_{add} \leftarrow MaxEval(A_N, n) - N$ // see text for $MaxEval$ definition
      8. for each non-empty subset $N'$ of $N_{add}$
         9. $P[N \cup N'] \leftarrow min_{cost}( SP( AND(N'), A, SP(AND(N), A \cup Attr(AND(N')), R) ),$
                                $P[N \cup N'] )$
10. for each child $n'$ of $n$
   11. for each subset $N'$ of $n.children$, containing $n'$
      12. if there is no $N''$ such that $N' \subseteq N''$ and $P[N'']$ is a pure plan then
         13. $P[N'] \leftarrow min_{cost}( SP(AND(N' - \{n'\}), A, IPG(n', A \cup Attr(AND(N' - \{n'\})), R)),$
                            $P[N'])$

// Step 2: Solve the MCSC problem and combine the sub-plans
14. Prune all dominated sub-plans from $P$
15. $E \leftarrow$ all subsets $N$ of $n.children$ such that $P[N] \neq \phi$
16. for each non-empty subset $SC$ of $E$ that evaluates all of $n$'s 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 $plan_{impure}$ // 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 non-empty subset N of n's children and checks if the query $SP(AND(N), A, R)$ is supported by R. As illustrated in Example 5.2, 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 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. This is 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 to evaluate 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. These sub-plans are then combined using a mediator intersection to obtain the best impure plan for $SP(n, A, R)$ (Lines 15–20). Before we solve the MCSC problem, we employ pruning rule pr_3 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.

6 Performance Evaluation

In this section, we study the performance of *GenCompact*. We demonstrate that it not only improves significantly on *GenModular* but also has good absolute performance in representative scenarios. We also study the quality of plans generated by *GenCompact* by comparing against existing systems (e.g., [25,12]).

The performance of *GenModular* and *GenCompact* strongly depends on the rewrite rules they employ. For our study, we consider a version of *GenModular* that uses the commutativity, associativity and copy rules, and a version of *GenCompact* that uses no rules. Since commutativity, associativity and copy rules are implicitly "built-into" *GenCompact*, these two versions will find the same best plan.

6.1 Comparison of *GenCompact* and *GenModular*

We compare the two schemes in terms of "total work," defined as $T \cdot W$, where T is the number of CTs produced by the respective rewrite modules, and W is the worst-case complexity of the plan generation for each CT. For computing plan generation complexity, we focus on two costs: the cost of parsing (calls to the $Check$ function), and the cost of the *EPG* and *IPG* calls.

For a given target query, the version of *GenCompact* under study works on a single CT, while *GenModular* must consider many CTs. At the same time, the cost of processing a CT in *GenCompact* is much higher than that in *GenModular*.

We focus on a scenario that results in a comprehensive total work ratio. In this scenario, each target query condition tree (denoted *ct*) is a balanced F-ary tree. That is, each non-leaf node has F children. We use H to denote the height of *ct* and $|CT|$ to denote the number of nodes in *ct*. Finally, C_q denotes the selection condition of the target query q and A_q denotes the projected attributes of q.

Ratio W_{gm}/W_{gc}: For *GenModular*, the worst-case estimate for work done per CT is

$$W_{gm} = O(|CT|^2 + |CT| \cdot 2^{|Attr(C_q) - A_q|} \cdot 2^F). \tag{6}$$

The first term represents the parsing costs incurred in the mark module. That module processes each of the $|CT|$ nodes, and for each parses the subtree rooted there in $O(|CT|)$ time.

The second term represents the cost of the *EPG* calls. Procedure *EPG* is called on each node at least once, accounting for the $|CT|$ factor. However, a given node can be called multiple times, possibly with different sets of requested attributes. We assume that if an *EPG* call is repeated for the same node and attributes, it is not performed twice. That is, we assume results of *EPG* invocations are cached. The middle factor (of the second term) accounts for how many times *EPG* can be called on each node with *different* attributes. Since the A_q attributes are always included any time *EPG* is called (any sub-query needs to retrieve them since they are needed in the final answer), this only leaves us with $|Attr(C_q) - A_q|$ attributes to include or not include in each call. The third factor (of the second term) accounts for the actual cost of each *EPG* invocation. Within an *EPG* call, we may have to invoke other *EPG* instances a maximum of 2^F times, once for each possible subset of the children of the node. (Notice that this factor only counts the invocation cost; the actual cost of each *EPG* call is counted when we consider the called node.)

For *GenCompact*, the worst-case estimate for work done per CT is

$$W_{gc} = O(|CT|^2 \cdot 2^F + |CT| \cdot 2^{|Attr(C_q)-A_q|} \cdot 2^{2^F}). \tag{7}$$

The first term of Equation (7) is similar to the first term of Equation (6). However, in the case of *GenCompact*, each subset of children nodes of a non-leaf node constitutes a condition that needs to be parsed. Hence, for each CT node, *GenCompact* makes about $2^F - 1$ additional parser calls. The second term of Equation (7) is similar to the second term of Equation (6). However, each *IPG* invocation may take $O(2^{2^F})$ time because it needs to solve the MCSC problem using an exhaustive algorithm.

With Equations (6) and (7) we can compute the work ratio as

$$\frac{W_{gm}}{W_{gc}} = \frac{1}{2^{2^F}}. \tag{8}$$

Ratio T_{gm}/T_{gc}: Since we assumed that *GenCompact* processes a single CT, T_{gc} is equal to one. On the other hand, for each non-leaf node n of the CT, *GenModular* examines all possible 2^{2^F} combinations of children node sets. Intuitively, 2^{2^F} different subtrees rooted at n are examined. Secondly, *GenModular* examines all possible orders of n's children nodes. It follows that the number of subtrees examined for each non-leaf node n is at least $F! \cdot 2^{2^F}$. So far, we have only considered one non-leaf node. Since the decision at each non-leaf node is independent of other decisions and there are $\frac{F^H-1}{F-1}$ non-leaf nodes, T_{gm} is given by

$$T_{gm} = (F! \cdot 2^{2^F})^{\frac{F^H-1}{F-1}}. \tag{9}$$

Summary: By combining Equations (8) and (9) we obtain the total work ratio:

$$\frac{TotalWork_{gm}}{TotalWork_{gc}} = (F!)^{\frac{F^H-1}{F-1}} \cdot (2^{2^F})^{\frac{F^H-1}{F-1}-1}. \tag{10}$$

This ratio clearly shows that *GenCompact* is much more efficient than *GenModular*. For instance, a simple DNF target query condition with 3 terms and 3 atomic conditions per term leads to a ratio of 2^{24}.

6.2 Efficiency of *GenCompact*

It is important to note that the parameters in Equation (7) are often small. For instance, a sizable DNF condition with 3 terms, each term having 3 atomic conditions, has a $|CT|$ equal to 13 and an F equal to 3. Given these numbers, *GenCompact* makes at most 104 parser calls (i.e., $|CT| \cdot 2^F$) each with an input size of at most 13 nodes (i.e., $|CT|$). Similarly, each MCSC problem is solved for at most 8 sub-plans (i.e, 2^F). Furthermore, if the conditions use few attributes beyond those that are projected in the final result, then $|Attr(C_q) - A_q|$ is small. On a modern computer, these *GenCompact* costs seem reasonable, given that this can lead to an Internet query that is feasible and very efficient (as we show in Section 6.3).

More importantly, keep in mind that Equation (7) gives a very pessimistic worst-case estimate of *GenCompact*'s performance. In particular, two factors can significantly reduce the actual costs. First of all, if the sources have limited capabilities, only a small subset of the potential 2^F sub-plans is actually feasible. Secondly, Equation (7) does not take into account all the pruning techniques used by *IPG*.

To estimate the average-case complexity of *GenCompact*, we implemented the *GenCompact* scheme and measured the work performed in processing queries in practice. The cost of *GenCompact* is dominated by the parser calls (actually, calls to the *Check* function) it makes and the MCSC problems it solves.

In one scenario, we examined four sources (the Library of Congress web site, the Amazon.com Internet bookstore, the Junglee Corporation web site, and a source that processes conjunctive queries). For each source, we hand-formulated a simple query (with 2 or 3 atomic conditions) and a complex query (with 5 to 10 atomic conditions) that "make sense" for that source. Appendix A explains these queries in more detail.

Figure 7 shows the results of our experiment. In that, Q_avg refers to the average input size to the set of MCSC problems solved for the query, while Q_max refers to the maximum input size to the MCSC problems solved. Over all sources and queries studied, the average number of parser calls made per query was 24.5, the average size (number of nodes) of the condition tree parsed was 3.9, the average number of times the MCSC problems solved was 1.3 and the average input size (number of sub-plans) for the MCSC problems was 1.2. We also observed that the maximum input size to any MCSC problem over all sources and queries was 3. These numbers indicate that the work that *GenCompact* performs is very reasonable, at least for these "human-generated" queries. The effort is most likely insignificant when compared with that involved in executing the queries over the Internet.

In a second scenario we considered a larger number of queries, but the queries were constructed randomly. We examined random queries of three different sizes: small (2 or 3 atomic conditions), medium (about 5 atomic conditions), and large (about 10 atomic conditions). The larger queries may model situations where a front-end tool assists a human in generating queries.

We considered the same set of Internet sources as before, and produced 10,000 random queries for each query size and source. Figure 8 shows the results of our experiment. Over all the sources, for small query condition expressions, the average number of parser calls was 7.2, the average size

Source	Query	# of Parser Calls	$\|CT\|_{avg}$	# of MCSC Problems	Q_{avg}	Q_{max}
Amazon	q_a	1	4	0	0	0
(www.amazon.com)	q_b	14	3.14	2	2	3
Library of Congress	q_c	15	2.13	1	1	1
(lcweb.loc.gov)	q_d	16	3.12	2	1	1
Junglee	q_e	5	5.4	2	1.5	2
(www.junglee.com)	q_f	3	8.33	1	2	2
CQ	q_g	15	2.13	1	1	1
(conjunctive source)	q_h	127	3.50	1	1	1

Figure 7: Efficiency of *GenCompact* in Processing "Sensible" Queries (Actual queries are in Appendix)

of the condition tree parsed was 2.2, the average number of MCSC calls was 1.3 and the average size of the MCSC problem input was 1.4. For large expressions, the numbers were 85.8, 5.8, 4.2 and 1.6 respectively. Even though the number of parser calls increased noticeably, it is still quite reasonable given that the large queries have many atomic conditions (about 10).

In the random queries scenario, we observed that the *maximum* MCSC size was much larger (93) than the average for one of the sources (Library of Congress). We believe that it is because this source is very powerful and so it yields many feasible sub-plans for parts of the target query condition. In cases such as this, one may need to consider more efficient MCSC algorithms that solve the problem approximately. This, of course, may cause *GenCompact* to generate sub-optimal plans for the target queries. To understand this tradeoff, we conducted simple experiments involving a version of *GenCompact* that uses a very efficient greedy algorithm to solve the MCSC problems. We observed that not using an exhaustive algorithm has minimal effect on the cost of the query plans produced. The greedy *GenCompact* often produced plans that are within 3% of those produced by the exhaustive *GenCompact*.

6.3 Quality of *GenCompact* Plans

In this section, we compare the plans produced by *GenCompact* with the plans produced by the following two schemes:

1. *GenDNF*: The condition expression of the target query is transformed to DNF and a plan is generated independently for each term. The plan for each term evaluates as many of its atomic conditions as possible at the source, as in [25]. The plans for the terms are combined with a mediator union to arrive at the plan for the target query.
2. *GenCNF*: The target query condition expression is transformed to CNF and all the clauses that can be evaluated at the source are determined. All such clauses are evaluated at the source while the rest of the clauses are evaluated at the mediator.

Source	Type of Query	# of Parser Calls	$\|CT\|_{avg}$	# of MCSC Problems	Q_{avg}	Q_{max}
Amazon	small	7.6	2.1	1.4	1.2	3
(www.amazon.com)	medium	18.8	3.13	2.3	1.3	5
	large	87.3	5.7	4.6	1.4	7
Library of Congress	small	6.2	2.3	1.2	1.8	7
(lcweb.loc.gov)	medium	15.5	3.4	1.7	1.95	15
	large	80.8	6.1	3.4	2.2	93
Junglee	small	7.3	2.2	1.4	1.3	3
(www.junglee.com)	medium	18.3	3.2	2.3	1.4	5
	large	86.3	5.8	4.5	1.5	8
CQ	small	7.4	2.1	1.4	1.2	3
(conjunctive source)	medium	18.3	3.2	2.2	1.3	5
	large	88.7	5.8	4.4	1.4	8

Figure 8: Efficiency of *GenCompact* in Processing a Random Query

GenDNF and *GenCNF* represent the kinds of techniques used to process target queries in contemporary systems like [12,20,21,25]. Since the *GenCompact* version we studied in our experiments did not use the distributivity rewrite rule, it is possible for *GenDNF* and *GenCNF* to generate better query plans than *GenCompact*.

To compare the three schemes, we considered the same sources as in Section 6.2 and generated 30,000 random target queries (small, medium and large) for each of them. For each query and source, the plan generated by each scheme was determined, and its cost obtained using the cost model of Section 5.2. The average cost over all the queries was calculated for each scheme and compared against that of the other schemes.

Figure 9 compares the average cost of the query plans produced by each scheme for each source and query combination. Each row marks the "winning" scheme that attained the lowest average cost. The columns of the losing schemes show how much the winning scheme improved on the cost of the losing schemes. For example, the plans for small *Amazon* queries generated by *GenCompact* take an estimated 40% of the time taken by the plans generated by *GenDNF*. *GenCompact* produced better plans in almost all the cases. On average, the cost of *GenCompact* plans was about 50% of that of *GenDNF* plans, and about 40% of that of *GenCNF* plans. These numbers only covered cases where all the three schemes produced a feasible plan. In our experiments, we observed that *GenCompact* found feasible plans in many more cases than *GenDNF* and *GenCNF*.

7 Discussion

So far, we have seen how *GenCompact* efficiently generates good feasible plans for selection queries. It uses a specific cost model suitable for the Internet context, along with a powerful source capability description language that can describe selection query capabilities of Internet sources very well. In

Source	Query Type	*GenCompact*	*GenDNF*	*GenCNF*
Amazon	small	winner	40%	46%
(www.amazon.com)	medium	winner	60%	54%
	large	winner	90%	54%
Library of Congress	small	winner	30%	69%
(lcweb.loc.gov)	medium	3.2%	winner	86%
	large	0.6%	winner	91%
Junglee	small	winner	39%	41%
(www.junglee.com)	medium	winner	67%	52%
	large	winner	91%	47%
CQ	small	winner	13%	64%
(conjunctive source)	medium	winner	37%	71%
	large	winner	73%	59%

Figure 9: Comparative Study

this section, we will briefly discuss how *GenCompact* can help generate source capability sensitive query plans for complex queries involving unions and joins. We will also describe how it can adapt to other source capability description languages and other cost models. We also consider the issue of doing efficient selection query processing involving expensive attributes.

Complex Queries. The ability of *GenCompact* to generate source capability sensitive query plans for selection queries can be very useful in dealing with more complex queries. One way in which *GenCompact* can help in generating good feasible plans for complex queries is as follows. Given a complex query, we can first generate plans for the complex query assuming that all the constituent selection queries on the data sources are feasible. This can be done using conventional optimization techniques. Then *GenCompact* can be invoked on each of the constituent selection queries in this query plan. The best feasible plans for the constituent selection queries are then combined with the earlier plan for the complex query to arrive at the best feasible plan for the complex query.

The above simple approach to using *GenCompact* in generating efficient feasible plans for complex queries has the advantage of yielding a very modular solution. This can be a big win when dealing with complex software components like query optimizers. We do note however, that this simple approach may not be able to generate feasible plans in some situations involving join operations. This is true only if the mediator considers the option of using *parameterized* ([21]) or *bind* ([12]) joins. [21,25] have specifically dealt with source capability sensitive query planning for join queries. The techniques developed there need to be combined with *GenCompact*, in order to provide more powerful feasible plan generation for complex queries involving joins, along with the select, project and union operations.

Other Capability Description Languages. Even though we chose a specific source capability description language in this paper, the dependency of *GenCompact* on this language is not very strong. All that it assumes of the capability description language is that there is a way to implement the *Check* function (see Section 3). It also requires that this function can automatically accommodate commutative transformations of the condition expression passed to it. *GenCompact* can easily adapt to other languages that describe source capabilities ([25,21]). For example, in the case of RQDL ([25]), it is very easy to implement a *Check* function satisfying the assumptions of *GenCompact*.

Other Cost Models. The *GenCompact* scheme of Section 5 employed a specific cost model that is suitable for the Internet context. This cost model led to a set of pruning rules that are used by *GenCompact* to achieve efficient plan space exploration. We believe that *GenCompact* can easily adapt to many other cost models. First of all, some or all of the pruning rules used by *GenCompact* may well be valid in many cost models. In this case, *GenCompact* continues to be very efficient in finding good feasible plans for the target queries. If the pruning rules are not strictly valid in a given cost model, *GenCompact* may choose to not employ those pruning rules. In this case, it still finds good feasible plans, even though its plan generation process may be more expensive. Finally, one may choose to employ some of the pruning rules in *GenCompact* even though they are not valid in a cost model under consideration. It may be the case that the pruning rules still provide good heuristics that lead *GenCompact* to find good feasible plans in most situations, while maintaining an efficient plan generation process.

Expensive Attributes. When expensive attributes are involved in selection queries, special care should be taken in executing source queries projecting these attributes. For instance, a target query may specify large attributes like images in its "projection list". It may not be prudent to import these expensive attributes of a data object to the mediator when the data object still needs to go through further filtering (with respect to the target query condition). One simple approach that may often yield good plans in these situations is to adopt a two-phase query processing strategy. In the first phase, only the key attributes of the answer objects are obtained. That is, we come up with a new target query based on the original target query by removing expensive attributes that are not key attributes. For the new target query *GenCompact* can be employed to arrive at an efficient feasible plan. In the second phase, we can send the key attributes to the data sources and obtain the expensive attributes of the answer objects. Note that this strategy works well only if the data objects at the sources have key attributes and if these attributes are not expensive themselves.

8 Conclusion

In this paper we studied the generation of efficient feasible plans for selection queries in the Internet context. We employed a simple language based on context free grammars to describe the query capabilities of the data sources. We presented an efficient scheme called *GenCompact* that generates good feasible plans. We studied the time complexity and plan quality of *GenCompact* using sample queries on a few interesting Internet sources.

References

[1] http://www.amazon.com/exec/obidos/ats-query-page/8600-3554296-148056

[2] http://autoconnect.net/FindVehicle/FVEntryPage.cfm

[3] http://shop.barnesandnoble.com/BookSearch/search.asp?userid=5QE80QDZ36

[4] http://www.junglee.com/shop/index.html

[5] http://lcweb.loc.gov/

[6] http://www.best.com/ sccref/catalog.html

[7] http://www.secapl.com/

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

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

[10] 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.

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

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

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

[14] 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.

[15] J. Hellerstein and M. Stonebraker. Predicate Migration: Optimizing Queries with Expensive Predicates. In *Proc. SIGMOD Conference*, 1993.

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

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

[18] A. Kemper, G. Moerkotte, K. Peithner and M. Steinbrunn. Optimizing Disjunctive Queries with Expensive Predicates. In *Proc. SIGMOD Conference*, 1994.

[19] D. Konopnicki and O. Shmueli. W3QS: A Query System for the World-wide Web. In *Proc. VLDB Conference*, 1995.

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

[21] C. Li, R. Yerneni, V. Vassalos, Y. Papakonstantinou, H. Garcia-Molina and J. Ullman. Capability Based Mediation in TSIMMIS. In *Proc. ACM SIGMOD Conference*, 1998.

[22] W. Litwin, L. Mark and N. Rossopoulos. Interoperability of Multiple Autonomous Databases. In *ACM Computing Surveys*, Vol. 22, No. 3, 267-293.

[23] W. Meng and C. Yu. Query Processing in Multidatabase Systems. In *Modern Database Systems.* W. Kim, ed. Addison Wesley, 1995.

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

[25] Y. Papakonstantinou, A. Gupta and L. Haas. Capabilities-based Query Rewriting in Mediator Systems. In *Proc. PDIS Conference*, 1996.

[26] 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.

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

[28] A. Sheth and L. Larson. Federated Database Systems for Managing Distributed, Heterogeneous and Autonomous Databases. In *ACM Computing Surveys*, Vol. 22, No. 3, 183-236.

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

[30] M. Stonebraker. The Ingres Papers. Addison Wesley, Reading (MA), 1986.

[31] A. Tomasic, L. Raschid and P. Valduriez. Techniques for Scaling Access to Distributed Heterogeneous Databases in DISCO. In *Proc. Int. Conf. on Distributed Computing Systems*, 1996.

[32] M. Tork Roth and P. Schwarz. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Sources. In *Proc. VLDB Conference*, 1997.

[33] V. Vassalos and Y. Papakonstantinou. Describing and Using the Query Capabilities of Heterogeneous Sources. In *Proc. VLDB Conference*, 1997.

A Details of Study

In this Appendix, we give the details of the study summarized in Figure 7, Section 6.

A.1 Source `http://www.amazon.com`

Consider the search web page of Amazon Inc. as our Internet source R ([1]) which only supports a conjunction of atomic conditions on attributes *author*, *title*, and *subject*. Also, the source always returns all three attributes together with the *price* attribute.

Let us suppose we are given the simple target query

$$q_a = \pi_{price}(\sigma_{c_1(author) \wedge c_2(title) \wedge c_3(subject)}(R)).$$

(We use $c_i(attr)$ to denote an atomic condition on attribute *attr*.) Since the selection condition can be evaluated at R, it should be evident that *GenCompact* obtains the pure plan without much effort as shown in the first row of Figure 7.

Let us suppose we are given the more complex target query

$$q_b = \pi_{title}(\sigma_{c_1(author) \wedge c_2(price) \wedge (c_3(subject) \vee c_4(subject) \vee c_5(subject))}(R)).$$

Let us denote the root node representing the entire selection condition as n_0 and the node representing the condition $(c_3(subject) \vee c_4(subject) \vee c_5(subject))$ as n_1. When *GenCompact* processes n_1, it finds three feasible sub-plans after making 7 parser calls with an average input condition size of about 2 nodes. *GenCompact* then comes up with a plan that evaluates $Cond(n_1)$ by solving the MCSC problem with an input size of 3 sub-plans. When *GenCompact* processes n_0, it makes 7 additional parser calls with an average input condition size of 4 nodes. It also finds 8 sub-plans that can help evaluate $Cond(n_0)$ but it prunes all but one sub-plan! For instance, it prunes the sub-plan $SP(c_2(price), title, SP(c_1(author), \{title, author, subject, price\}, R))$ since source R *always* exports all four attributes. Hence, the sub-plan that also locally evaluates the atomic conditions on *subject* in addition to $c_2(price)$ dominates the previous sub-plan. A summary of the processing involved in obtaining a plan for q_b is shown in the second row of Figure 7.

A.2 Source `http://lcweb.loc.gov`

Let us consider the advanced search web page of the Library of Congress ([5]) as our source R. The source is powerful enough to evaluate atomic conditions on any of its 8 attributes (e.g., *title*, *subject*, *corp*, *name*, *series*) connected arbitrarily by $\wedge$'s or $\vee$'s. Also, the source always exports all 8 attributes. However, the source limits the selection conditions to at most 3 atomic conditions.

Let us consider the following queries:

$$\begin{aligned} q_c &= \pi_{name,isbn,issn}(\sigma_{c_1(name) \wedge c_2(corp) \wedge c_3(title) \wedge c_4(subject)}(R)) \\ q_d &= \pi_{name,isbn,issn}(\sigma_{c_1(name) \wedge c_2(corp) \wedge c_3(title) \wedge (c_4(subject) \vee c_5(subject))}(R)) \end{aligned}$$

GenCompact finds a query plan for q_c by making 15 parser calls with an average condition size of about 2 nodes. By making these parser calls, *GenCompact* finds 56 sub-plans but prunes all but one of them! Hence, it solves the MCSC problem with an input size of just 1 sub-plan.

GenCompact finds a query plan for target query q_d in a similar fashion but makes an additional parser call when the node n representing condition $(c_4(subject) \vee c_5(subject))$ is processed. *Gen-Compact* also solves one more instance of the MCSC problem when it processes node n but with an input size of only 1 sub-plan. The third and fourth rows of Figure 7 summarize the processing involved in obtaining the query plans for q_c and q_d.

A.3 Source `http://www.junglee.com`

Let us consider the "apartment finder" web page of Junglee Corp. ([4]) as our source R. The source accepts a list of locations that the apartment should be in (i.e., a disjunction of atomic conditions on loc). In conjunction, it also accepts a specification on the number of bedrooms (denoted $c(bedroom)$) and a cap on the price (denoted $c(price)$).

Let us consider the following query:

$$q_e \quad = \quad \pi_{address,price}(\sigma_{(c_1(loc) \vee c_2(loc)) \ \wedge \ ((c_3(bedroom) \wedge c_4(price)) \vee (c_5(bedroom) \wedge c_6(price)))}(R)).$$

When *GenCompact* processes the root node n_0 of the CT, it makes 3 parser calls with input sizes of 11, 3 and 7 nodes. Using these parser calls, *GenCompact* identifies that the sub-plan $SP(c_1(loc) \vee c_2(loc), \{address, price, bedroom, price\}, R)$ is feasible. *GenCompact* also processes the node n_1 that represents the condition $(c_3(bedroom) \wedge c_4(price)) \vee (c_5(bedroom) \wedge c_6(price))$. *GenCompact* makes 2 additional parser calls both with input size of 3 nodes. The plan to evaluate $Cond(n_1)$ is found by solving the MCSC problem with an input size of 2 sub-plans. Finally, the plan to evaluate $Cond(n_0)$ is found by solving the MCSC problem with an input size of 1 sub-plan since all other sub-plans are pruned. The fifth row of Figure 7 summarizes the processing involved.

One application of the distributivity rule on the selection condition of q_e results in the following target query:

$$q_f \quad = \quad \pi_{address,price}(\sigma_{((c_1(loc) \vee c_2(loc)) \wedge c_3(bedroom) \wedge c_4(price)) \ \vee \ ((c_1(loc) \vee c_2(loc)) \wedge c_5(bedroom) \wedge c_6(price))}(R)).$$

GenCompact finds the best query plan obtainable from this target query using 3 parser calls with an average input size of about 8 nodes and by solving the MCSC problem once with an input size of 2 sub-plans (see sixth row, Figure 7). Due to lack of space, we do not give the details here.

A.4 Conjunctive Sources

Consider a hypothetical source that accepts a conjunction of at most 3 atomic conditions. Given a query q_g that is a conjunction of 4 atomic conditions, 15 parser calls $(2^4 - 1)$ identify 14 possible sub-plans. Only one of the sub-plans remain if the source exports all of its attributes. Otherwise, at most $\binom{4}{3}$ sub-plans remain after pruning dominated sub-plans. Similarly, a query g_h that is a conjunction of 8 atomic conditions can be processed in 127 parser calls $(2^8 - 1)$ resulting in either 1 or $\binom{8}{3}$ sub-plans depending on what the attributes the source exports.