Mind Your Vocabulary:
Query Mapping Across Heterogeneous Information Sources *

Chen-Chuan K. Chang
Electrical Engineering Department

Héctor García-Molina
Computer Science Department

Stanford University
Stanford, CA 94305-9040, USA
{kevin,hector}@db.stanford.edu

Abstract

In this paper we present a mechanism for translating *constraint queries*, *i.e.*, Boolean expressions of constraints, across heterogeneous information sources. Integrating such systems is difficult in part because they use a wide range of constraints as the vocabulary for formulating queries. We describe algorithms that apply user-provided mapping rules to translate query constraints into ones that are understood and supported in another context, *e.g.*, that use the proper operators and value formats. We show that the translated queries *minimally* subsume the original ones. Furthermore, the translated queries are also the most compact possible. Unlike other query mapping work, we effectively consider inter-dependencies among constraints, *i.e.*, we handle constraints that cannot be translated independently. Furthermore, when constraints are not fully supported, our framework explores relaxations (semantic rewritings) into the closest supported version. Our most sophisticated algorithm (Algorithm *TDQM*) does not blindly convert queries to DNF (which would be easier to translate, but expensive); instead it performs a top-down mapping of a query tree, and does local query structure conversion only when necessary.

1 Introduction

For seamless information access, mediation systems [1, 2] have to cope with the different data representations and search capabilities of sources. To mask the heterogeneity, a mediator presents a unified context to users. The mediator must translate queries from the unified context to the native contexts for source execution. This translation problem has become more critical now that the Internet and intranets have made available a wide variety of disparate sources, such as multimedia databases, web sources, legacy systems, and information retrieval (IR) systems. In this paper we show how to efficiently translate queries, taking into account differences in operators, data formats, and attribute names.

* This work was partially supported by NSF Grant IRI-9411306 and IIS-9811992.

Example 1: Suppose that a mediator integrates on-line bookstores to provide book information (such as the services provided by the web site `www.acses.com` and `shopping.yahoo.com`). In particular, the mediator exports an integrated view $Book$(title, ln, fn, ...) with attributes for title, author last name, first name, *etc*. To search for books, users specify constraints in their queries. Suppose that a user is looking for books by Tom Clancy, *i.e.*, the constraint query Q is [fn = "Tom"] $\wedge$ [ln = "Clancy"].

The mediator must then translate the query to search the underlying sources. For instance, consider source *Amazon* (at `www.amazon.com`). This source does not understand attribute ln and fn; instead, it supports the author attribute, which requires some particular name format. Thus, the translation for *Amazon* should be [author = "Clancy,Tom"].

In addition, let's consider source *Clbooks* (*i.e.*, Computer Literacy at `www.clbooks.com`). *Clbooks* also supports author but allows only operator contains (instead of equality) that searches any words in names. While Q is not fully expressible at *Clbooks*, we can come up with the mapping Q_c = [author contains Tom] $\wedge$ [author contains Clancy]. Strictly speaking, this translation is not equivalent; Q_c is in fact a relaxation of Q (*i.e.*, Q_c subsumes Q). For instance, "Tom,Clancy" and "Clancy,Joe Tom" match Q_c but not Q. Thus, the mediator needs to redo Q as a *filter* to remove the false positives returned from *Clbooks*. ■

We can view a query as a Boolean expression of constraints of the form [attr1 op value] or [attr1 op attr2]. These constraints constitute the "vocabulary" for the query, and must be translated to constraints understood by the target source. This constraint mapping must consider source capabilities, and thus is not symmetrical to data conversion (see Section 3). In general, we have to map attributes (*e.g.*, cost to price), convert data values (*e.g.*, 3 inches to 7.62 centimeters), and transform operators (*e.g.*, "=" to "contains").

It is also critical to note that query mapping is not simply a matter of translating each constraint separately. Some constraints can be inter-dependent and must be handled together. In general, constraint mapping is *many-to-many*. For instance, the query [car-type = "ford-taurus"] $\wedge$ [year = 1994] may yield [make = "ford"] $\wedge$ [model =

"taurus-94"] at the source. Without respecting *constraint dependencies*, a translation cannot guarantee the *minimal* mappings that are as selective as possible.

Example 2: Consider translating for *Amazon* the query $Q = C_1 \wedge C_2 = (f_1 \vee f_2) \wedge f_3$, where f_1 = [ln = "Clancy"], f_2 = [ln = "Klancy"], and f_3 = [fn = "Tom"]. Note that *Amazon* supports attribute author, of which the last name must be specified. (Thus, a name can be "Clancy,Tom", or simply "Clancy" if the first name is not known.)

If we ignore the potential dependencies between constraints or subqueries, and separately translate C_1 and C_2, we may obtain only a suboptimal mapping. To illustrate, let $\mathcal{S}(X)$ denote the mapping of query X. Separating C_1 and C_2 (as well as f_1 and f_2), we obtain the mapping Q_a = $\mathcal{S}(C_1) \wedge \mathcal{S}(C_2)$ = $[\mathcal{S}(f_1) \vee \mathcal{S}(f_2)] \wedge \mathcal{S}(f_3)$. Note that $\mathcal{S}(f_3) = \mathit{True}$ (*i.e.*, no constraint) because *Amazon* cannot impose constraints on the first name alone. Thus $Q_a = \mathcal{S}(f_1) \vee \mathcal{S}(f_2)$ = [author = "Clancy"] $\vee$ [author = "Klancy"].

Q_a is actually not "minimal"; it is not as selective as Q_b = [author = "Clancy,Tom"] $\vee$ [author = "Klancy,Tom"] (which is in fact the minimal mapping). Intuitively, conjuncts C_1 and C_2 are "interrelated" and not separable as they together decide the target constraints on author. ∎

To obtain good translations, we must rely on human expertise, *e.g.*, to tell us that two constraints are interrelated, or that some function needs to be applied to transform inches to centimeters. Thus, we provide a rule-based framework for codifying the necessary domain semantics. For instance, one rule may tell us that a constraint [ln = L] can be mapped to [author = A], where L and A are variables that stand for values. The rule then provides a human-written function to transform the last name L to the author name A. Another rule may tell us that the pair of constraints [fn = F] and [ln = L] can be mapped to [author = A], using a different function that now maps a first and last name into a combined string.

Furthermore, based on these rules, our challenge is to translate a full query, where different portions of the query may match different rules. For instance, consider the query $(f_1 \vee f_2) \wedge f_3 \wedge f_4$. We may have a rule for mapping $(f_3 \wedge f_4)$ and another for $(f_2 \wedge f_3)$. This latter rule can be applied if we rewrite the query. Which rule should we apply? When and how should we rewrite the query? If we have rules for $(f_3 \wedge f_4)$ and for f_3, f_4 alone, which rules should we apply?

This paper presents an efficient algorithm (called Algorithm *TDQM*) for mapping queries according to a set of user-provided rules. The algorithm guarantees an optimal mapping, in which a translated query will minimally subsume the original one. (We will formally define this concept later; informally it means that the translated query will not return unwanted answers that were possible to avoid with some better translation.) In addition, in most cases the algorithm produces the most "compact" translated query, *i.e.*, the query with the smallest parse tree, out of the possible translations. The algorithm does not blindly convert queries to DNF, which would be easier to translate, but expensive. Instead it performs a top-down mapping of a query tree, and does local query structure conversion only when necessary.

Many integration systems have dealt with source capabilities, *e.g.*, [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]. We discuss the related work in Section 3, but in summary the essential features that distinguish our work are:

- We address *dependencies* that exist among constraints or subqueries; as far as we know, no other translation frameworks respect dependencies for optimal mapping.
- We deal with *arbitrary* constraints; other systems typically push only simple equality constraints to sources.
- We perform systematic *semantic mapping* of constraints (with human-specified rules); most other systems only handle syntactic translation, and do not take advantage of relaxing an unsupported constraint semantically.
- We efficiently process *complex* queries (with conjunctions and disjunctions). Most other systems focus on simple conjunctive queries, or process complex queries in DNF, which is expensive in general.

This paper focuses on the constraint mapping problem, and does not consider other important translation issues, *e.g.*, the subsequent generation of physical query plans (many related efforts have addressed this issue). Note also that, while we handle complex queries, we currently do not consider negations. Furthermore, we discuss in reference [15, 16] the generation of effective filter queries (Example 1 illustrated why they were needed). Also, due to space limitations, we only consider constraints of the form [attr1 op value], and not ones of the form [attr1 op attr2] that may arise in a join query. The extensions to our approach for the join constraints are not extensive, and are discussed in [17].

We start by defining the constraint mapping problem and other fundamental notions. In Section 3 we review the related efforts. Section 4 describes the basic mapping mechanism for conjunctive queries. For complex queries, Section 5 discusses a framework based on the DNF of queries. Section 6 then presents Algorithm *TDQM* that does not require DNF. In Section 7 we discuss the separation of conjuncts, which is a critical foundation for Algorithm *TDQM*. Finally, Section 8 summarizes the complexity and correctness properties of Algorithm *TDQM*.

2 The Constraint Mapping Problem

We describe the constraint mapping problem in a common mediation architecture [1, 2] for integrating heterogeneous *sources*. In such systems, *wrappers* unify the source data models, and *mediators* interact with the wrappers to process queries transparently. A mediator exports integrated *mediator views* for users to formulate queries. Thus, a *user query* $\mathcal{U}$ over some views V_i has the form (in an SQL-like expression) **select** ... **from** $V_1, \ldots, V_h$ **where** C, or algebraically $\mathcal{U} = \sigma_C(V_1 \times \cdots \times V_h)$, where C is a Boolean expression of *constraints*. (The projection operation is omitted as it is irrelevant to our discussion.)

Note that we do not consider negation in this paper. A constraint is either a *selection* condition [V_i.attr1 op value], or a *join* condition [V_i.attr1 op V_j.attr2], where attr1 and attr2 are attributes of view V_i and V_j respectively.

In such mediation frameworks, a view is typically an SPJ query over some source relations plus possibly some *data conversion* functions; *e.g.*, view (title, ln, fn, review) might be a join of relation (title, review) from source T_1, (title, author) from T_2, and a function NameLnFn(author, ln, fn) for converting author to last and first names. We can model such a function as a *conceptual relation* with the tuples [author, ln, fn] that "satisfy" the function.

For source execution, the mediator must rewrite a user query in terms of the source relations. Thus, with view expansion, $\mathcal{U}$ will be rewritten to the following form in Eq. 1, where $\mathbf{R}_i$ is the cross-product of all the source relation instances that a particular source T_i contributes to any queried views, and $\mathbf{X}$ is the cross product of the relevant conceptual relations. We specifically refer to the selection condition Q as a *constraint query*: In most cases Q is simply the user-query condition C, but in addition Q can also include the constraints used in the view definitions.

$$\mathcal{U} = \sigma_Q(\mathbf{R}_1 \times \cdots \times \mathbf{R}_n \times \mathbf{X}) \quad (1)$$

Intuitively, the *constraint mapping* problem is to *push* as much as possible the constraint query to the sources. That is, the mapping translates Q from the mediator's *original context* to the *target context* at each source. Note that the constraints in Q are generally not readily executable across different contexts. First, there exists *schema* difference between the views and the sources: The conversion functions in $\mathbf{X}$ can present new attributes (*e.g.*, ln and fn that replace author) or change data representations. Second, there exists *capability* difference: Unless the mediator only allows the least common denominator of what the sources support, the constraints can be beyond the capabilities of some sources.

Thus, constraint mapping will find the mapping of Q for each source T_i, denoted $\mathcal{S}_i(Q)$, to retrieve the relevant subset of $\mathbf{R}_i$. The mediator then combines these source results, passes them through the conversion functions, and postprocesses with a *filter* query F consisting of the residue conditions not fully pushed to the sources, *i.e.*,

$$\mathcal{U} = \sigma_F[\sigma_{\mathcal{S}_1(Q)}(\mathbf{R}_1) \times \cdots \times \sigma_{\mathcal{S}_n(Q)}(\mathbf{R}_n) \times \mathbf{X}]. \quad (2)$$

Comparing Eq. 1 and Eq. 2, we obtain the essential property for a *correct* translation:

$$Q = F \wedge \mathcal{S}_1(Q) \wedge \cdots \wedge \mathcal{S}_n(Q) \quad (3)$$

Example 3: To illustrate the translation problem, let us consider a mediator for two sources. Suppose that source T_1 provides relation A(ti, au) for paper titles and authors and B(name, bib) for author names and their bibliography. Source T_2 has C(ln, fn, dept) for faculty last, first names, and departments. The mediator exports view fac(ln, fn, bib, dept) integrated from B and C, and view pub(ti, ln, fn) from A(ti, au).

Suppose that a user is looking for the papers written by some CS faculty interested in data mining. The constraint

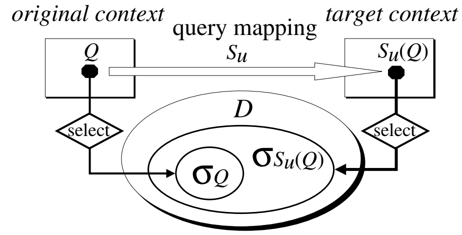

Figure 1: Conceptual illustration of query mapping.

query is Q = a:[fac.ln = pub.ln] $\wedge$ b:[fac.fn = pub.fn] $\wedge$ c:[fac.bib contains data(near)mining] $\wedge$ d:[fac.dept = cs]. Note that Q includes both selection and join constraints.

Let's first consider the mapping for source T_1, *i.e.*, for relations A and B. The join conditions $a \wedge b$ together map to x_1 : [A.au = B.name]. If source T_1 does not support the proximity operator near, rather than dropping constraint c, we can relax it to (x_2:[B.bib contains data] $\wedge$ x_3:[B.bib contains mining]) that requires only the occurrences of keywords. Lastly, constraint d maps to *True* (it can only be processed in T_2). Thus, $\mathcal{S}_1(Q) = x_1 \wedge x_2 \wedge x_3$.

We next perform the mapping for source T_2, which contributes relation C. All the constraints except d map to *True*. Suppose that T_2 uses department code 230 for CS, thus $\mathcal{S}_2(Q)$ = [C.dept = 230].

Finally, the filter query F is simply the constraint c (*i.e.*, $F = c$), the only contraint that is not fully realized at the underlying sources. Thus, $Q = F \wedge \mathcal{S}_1(Q) \wedge \mathcal{S}_2(Q)$. ■

Since we can perform the mappings for different sources separately (as Example 3 illustrated), we now focus on a particular source T_u as the translation *target* and discuss the requirements for $\mathcal{S}_u(Q)$: To begin with, $\mathcal{S}_u(Q)$ must be *expressible* in target T_u; *i.e.*, $\mathcal{S}_u(Q)$ contains only those constraints that T_u supports with its schema and capability. (Thus, $\mathcal{S}_u(Q)$ uses only the native vocabulary of T_u.)

Furthermore, $\mathcal{S}_u(Q)$ logically subsumes Q; note that we can rewrite Eq. 3 as $Q = F_u \wedge \mathcal{S}_u(Q)$ (where F_u is the conjunction of F and $\mathcal{S}_i(Q)$, $i \neq u$). For a relation D (in this case $D = \mathbf{R}_1 \times \cdots \times \mathbf{R}_n \times \mathbf{X}$), Q' *subsumes* Q if $\sigma_{Q'}(D) \supseteq \sigma_Q(D)$ regardless of the contents of D. If $\sigma_{Q'}(D) \supset \sigma_Q(D)$ for some instance of D, then Q' *properly subsumes* Q. Thus, when source T_u evaluates $\mathcal{S}_u(Q)$ on (the $\mathbf{R}_u$ part of) relation D, it will select a superset of what Q does. Figure 1 shows this subsumption relationship. The extra tuples selected by the translated query will be removed by the corresponding filter F_u. Finally, we want $\mathcal{S}_u(Q)$ to return as few extra tuples as possible; *i.e.*, $\mathcal{S}_u(Q)$ should be the *most selective* mapping. In this case we say that $\mathcal{S}_u(Q)$ *minimally subsumes* Q with respect to T_u. In Definition 1 we summarize these three requirements for constraint mapping.

Definition 1 (Minimal Subsuming Mapping): A mapping $\mathcal{S}_u(Q)$ is the *minimal subsuming mapping* of a constraint query Q *w.r.t.* the target context T_u, if **(1)** $\mathcal{S}_u(Q)$ is expressible in T_u, **(2)** $\mathcal{S}_u(Q)$ subsumes Q, and **(3)** $\mathcal{S}_u(Q)$ is minimal, *i.e.*, there is no query Q' such that (i) Q' satisfies 1 and 2, and (ii) $\mathcal{S}_u(Q)$ properly subsumes Q'. ■

To illustrate, recall that the mapping Q_a in Example 2 is not minimal. To see why, note that there exists another mapping Q_b (see Example 2) that is also expressible in the target context. Furthermore, Q_a properly subsumes Q_b.

This paper specifically discusses the algorithms for mapping a constraint query Q. Note that from now on we will simply refer to such Q as a *query* (not to be confused with a full *user query* $\mathcal{U}$). Also, we write the mapping as $\mathcal{S}(Q)$ (without a subscript) when the target source is clear as in Example 2. In addition, due to space limitations, we only consider selection constraints (of the form [attr1 op value]) in this paper. Our framework can handle join constraints (of the form [attr1 op attr2]) as well. We discuss the extensions in an extended technical report [17].

3 Related Work

While information integration has long been an active research area [1, 2, 18], the constraint mapping problem we study in this paper has not been addressed throughly. Many integration systems have dealt with source capabilities, *e.g.*, Information Manifold [3, 4], TSIMMIS [5, 6], Infomaster [7, 8], Garlic [9, 10], DISCO [11], and others [12, 13, 14]. Our work complements the existing efforts. We specifically address the semantic mapping of constraints, or analogously the translation of vocabulary. In contrast, other efforts have mainly focused on generating query plans that observe the native *grammar* restrictions (such as allowing conjunctions of two constraints, disallowing disjunctions, *etc.*).

First, many integration systems (TSIMMIS, Garlic, and DISCO) essentially follow the mediator-views approach as Section 2 discusses. For query translation, their mediators first perform *view expansion* to form logical plans, and then their wrappers generate physical plans with *capability-based rewriting*. They do process constraints, but often with simplistic assumptions. As mentioned in Section 1, the essential features that distinguish our work are:

- We address *dependencies* that exist among constraints or subqueries. Note that such dependencies can be quite common in practice because heterogeneous sources may use different attributes to structure the same information (*i.e.*, they may not have matching schemas), as we illustrated in Section 1.

  We are not aware of other translation frameworks that respect dependencies for optimal mapping. Other systems implicitly assume one-to-one mapping of constraints, which leads to suboptimal solutions as Example 2 illustrated. In particular, they can violate constraint dependencies when generating physical plans. For instance, Garlic processes complex queries in CNF and is not aware of dependencies. Some systems use grammar-like, rule-based languages (*e.g.*, QDTL [6], RQDL [19], CFG [12], ODL [11]) to describe acceptable query templates and the associated translations. However, these capability-description frameworks focus on the grammatic structure of queries. In particular, their rules do not encode and respect constraint dependencies, unlike ours (see Section 4).
- We deal with *arbitrary* constraints. Other systems (*e.g.*, [5]) that rely on mediator view expansion push to sources only simple equality constraints (of the form [attr = value]), *i.e.*, attribute *bindings* to exact values. (This masks the capabilities of the sources, because they may be able to process more sophisticated constraints.) Thus, the problem of constraint mapping is simplified to propagating bindings (such as from [ln = "Clancy"] $\wedge$ [fn = "Tom"] to [author = "Clancy,Tom"]). This propagation can use the same mechanism as data value conversion in view definition (as Section 2 discussed). For instance, the bindings on ln and fn can be mapped to author via a function `LnFnName` that is an inverse of the conversion function `NameLnFn` used in defining the views.

  However, constraint mapping is in general not symmetrical to data conversion: Unlike data values, queries can specify constraints that are partial (*e.g.*, giving only ln) and inexact (*i.e.*, non-equality, *e.g.*, [ln sounds-like "Klancy"]). Moreover, constraint mapping must also map operators to respect source capabilities. For instance, in Example 3, the mapping from `data`(near)`mining` to `data`($\wedge$)`mining` has nothing to do with data conversion.
- We perform systematic *semantic mapping* of constraints (with human-specified rules); most other systems do not take advantage of relaxing an unsupported constraint semantically. The wrappers of these systems simply translate a constraint syntacticly (*e.g.*, from [ln = "Klancy"] to the native command "`lookup -ln Klancy`") if supported, or else drop it *entirely*. Instead, semantic rewriting would explore to relax an unsupported constraint into a closest supported version (such as replacing near with $\wedge$ in Example 3).
- We efficiently process *complex* queries. Most other systems focus on simple conjunctive queries, or process complex queries in DNF, which is expensive in general. In contrast, our algorithms do not assume DNF (Section 6).

In addition, the second category of integration efforts adopts the *answering-queries-using-views* approach (*e.g.*, [3, 4, 7, 8, 13, 14]). This approach assumes a *world view* of global relations and global constraints, in which queries and sources can be described. However, the related efforts have not tackled how to localize this "global vocabulary" (*i.e.*, the world view). Thus, our work complements these efforts.

4 Simple-Conjunction Queries

Query translation must rely on human expertise. In this section we present a rule-based scheme that codifies such expertise. The scheme relies on rules to indicate what groups of constraints need to be mapped as a unit, and what user-provided functions must be executed to actually transform values (*e.g.*, to change the units or encoding of values). As we will see, the human-provided rules only

Original Query	Target Query for *Amazon*
$\hat{Q}_1 = f_l \wedge f_{t1} \wedge f_y \wedge f_m \wedge f_k$	$S_1 = a_a \wedge a_{t1} \wedge a_d \wedge (a_{t2} \vee a_{s1})$
f_l: [ln = "Smith"]	a_a: [author = "Smith"]
f_{t1}: [ti contains java(near)jdk]	a_{t1}: [ti-word contains java(∧)jdk]
f_y: [pyear = 1997]	a_d: [pdate during May/97]
f_m: [pmonth = 5]	a_{t2}: [ti-word contains www]
f_k: [kwd contains www]	a_{s1}: [subject-word contains www]
$\hat{Q}_2 = f_p \wedge f_{t2} \wedge f_c \wedge f_i$	$S_2 = a_p \wedge a_{t3} \wedge a_{s2} \wedge a_i$
f_p: [publisher = "oreilly"]	a_p: [publisher = "oreilly"]
f_{t2}: [ti = "jdk for java"]	a_{t3}: [title starts "jdk for java"]
f_c: [category = "D.3"]	a_{s2}: [subject = "programming"]
f_i: [id-no = "081815181Y"]	a_i: [isbn = "081815181Y"]

Figure 2: Mapping simple-conjunction queries.

specify how to translate the smallest grouping of basic constraints, *e.g.*, a pair of constraints that must be considered together for proper translation. The translation of full queries is then performed by a query translation algorithm, which relies on the rules to transform the basic constraints involved. In this section we describe the basic translation rules, and we discuss an algorithm that can translate any simple conjunctive query. In later sections we then present algorithms that can handle general Boolean queries. Our rule specifications are based on rules we developed earlier for data translation [20]. Here we adapt this framework for query translation. Please refer to [20] for a more formal definition of the rule framework.

Given a query $\hat{Q}$ as a conjunction of constraints in the original context, our goal is to find its minimal subsuming mapping $\mathcal{S}(\hat{Q})$ in the target context. (To stress that the query is conjunctive, we write it as $\hat{Q}$.) Our framework in [20] translates data (*i.e.*, attribute-value pairs) as conjunctive equality-constraints. This section briefly summarizes the extended framework that allows arbitrary constraints. In particular, we illustrate the mappings for target *Amazon*[1] from the original context of a mediator. For example, Figure 2 shows two original queries $\hat{Q}_1$ and $\hat{Q}_2$ translated for *Amazon* to S_1 and S_2 respectively. Note that we designate the original constraints with f_α and the target constraints a_β respectively, where α and β are some descriptive strings.

In translation, our framework first maps individual constraints according to a human-specified *mapping specification*, and then formulates the mapping of the whole original query. The mapping specification for a particular target is a set of *mapping rules*, *e.g.*, Figure 3 lists the rules K_{Amazon} for target *Amazon*.

A rule matches a set of (conjunctive) constraints and specifies its translation, similar to pattern matching in, *e.g.*, Yacc. As Figure 3 shows, the head (left hand side) of a rule consists of constraint *patterns* and *conditions* to match the original constraints. The tail (to the right of $\mapsto$) consists of *functions* for converting value formats and an *emit*: clause that specifies the target query.

[1]We assume a target context based on the "power search" interface at `www.amazon.com`, with slight changes for the purpose of illustration.

$\mathcal{R}_1$) [A1 O X]; SimpleMapping(A1) $\mapsto$
A2 = AttrNameMapping(A1); *emit*:[A2 O X]
$\mathcal{R}_2$) [ln = L]; [fn = F] $\mapsto$ A = LnFnToName(L, F);
emit:[author = A]
$\mathcal{R}_3$) [ln = L] $\mapsto$ *emit*:[author = L]
$\mathcal{R}_4$) [ti contains P1] $\mapsto$ P2 = RewriteTextPat(P1);
emit:[ti-word contains P2]
$\mathcal{R}_5$) [ti = T] $\mapsto$ *emit*:[title starts T]
$\mathcal{R}_6$) [pyear = Y1]; [pmonth = M1] $\mapsto$
Y2 = NormYear(Y1); M2 = NormMonth(M1);
D = MonthYearToDate(Y2, M2); *emit*:[pdate during D]
$\mathcal{R}_7$) [pyear = Y1] $\mapsto$ Y2 = NormYear(Y1);
emit:[pdate during Y2]
$\mathcal{R}_8$) [kwd contains K1] $\mapsto$ K2 = RewriteTextPat(K1);
emit:[ti-word contains K2]∨[subject-word contains K2]
$\mathcal{R}_9$) [category = C] $\mapsto$ S = MapCategoryTerms(C);
emit:[subject = S]

Figure 3: Mapping rules K_{Amazon} for *Amazon*.

For example, rule $\mathcal{R}_4$ in K_{Amazon} (Figure 3) maps a contains constraint on ti to one on attribute ti-word (*e.g.*, from f_{t1} to a_{t1} in Figure 2). When pattern [ti contains P1] matches a constraint (*e.g.*, f_{t1}), the variable P1 (in capitalized symbols) is *bound* to the corresponding constant, *i.e.*, P1 = "java(near)jdk". The matching of the head will fire the actions in the tail. In particular, it calls upon function RewriteTextPat to rewrite the text pattern P1. As *Amazon* does not support near, P1 is rewritten to P2 = "java(∧)jdk". (For instance, reference [21] describes a general procedure for translating such IR predicates.) As we mentioned, the functions (as well as the conditions in the head) are supplied externally, and in principle can be written in any programming languages. Finally, the *emission* (*i.e.*, the *emit*: clause) of the rule outputs the mapping as [ti-word contains P2] (*i.e.*, a_{t1} in Figure 2).

A rule can use conditions (*i.e.*, predicate functions) to restrict the matchings. For instance, while the pattern in $\mathcal{R}_1$ can match any constraint, condition SimpleMapping tests if A1 is bound to a "simple" attribute that requires only name mapping (with function AttrNameMapping) such as attributes publisher and id-no in Figure 2.

As illustrated, the evaluation of a rule finds the matching constraints and computes the emissions. Given a simple-conjunction $\hat{Q}$, a *matching* of a rule $\mathcal{R}$ is a subset of (the constraints in) $\hat{Q}$ that together satisfies the head of $\mathcal{R}$. A rule can have multiple matchings or none; we denote the set of all the matchings of rule $\mathcal{R}$ for query $\hat{Q}$ by $\mathcal{M}(\hat{Q}, \mathcal{R})$. For instance, consider $\mathcal{R}_1$ and assume that SimpleMapping(A1) holds only for attributes id-no and publisher. Referring to Figure 2, we get two matchings for $\hat{Q}_2$ (*i.e.*, $\mathcal{M}(\hat{Q}_2, \mathcal{R}_1) = \{\{f_p\}, \{f_i\}\}$) but none for $\hat{Q}_1$ (*i.e.*, $\mathcal{M}(\hat{Q}_1, \mathcal{R}_1) = \phi$). Moreover, a matching can have multiple constraints. For example, constraints f_y and f_m in $\hat{Q}_1$ together match $\mathcal{R}_6$, *i.e.*, $\mathcal{M}(\hat{Q}_1, \mathcal{R}_6) = \{\{f_y, f_m\}\}$. Furthermore, since constraint mapping is generally many-to-many, an emission can be a complex query (rather than a single constraint).

For instance, rule $\mathcal{R}_8$ produces the disjunctive constraints on ti-word and subject-word, assuming *Amazon* does not explicitly support a kwd attribute (for keywords).

Since the mapping rules just described are the critical basis of our translation framework, they must observe some requirements. We in fact assume that the human experts only give *sound* rules. First, the emission of a rule is *by definition* the minimal subsuming mapping of the corresponding matching. For instance, because for the matching $\{f_y,\ f_m\}$ rule $\mathcal{R}_6$ emits [pdate during May/97] (shown as a_d in S_1, Figure 2), we know that $\mathcal{S}(f_y \wedge f_m) = a_d$, if $\mathcal{R}_6$ is sound.

Furthermore, the matchings of a rule must be *inseparable*, *i.e.*, a rule should handle only those truly-dependent constraints. In other words, the mapping rules effectively encode constraint dependencies. For instance, for $\mathcal{R}_6$ the matching $\{f_y,\ f_m\}$ is indeed inseparable. Separating f_y and f_m would only result in a suboptimal mapping: Since *Amazon* requires that the year be specified in a pdate constraint, there is no mapping for only a month, *i.e.*, $\mathcal{S}(f_m) = \mathit{True}$. Thus, $\mathcal{S}(f_y) \wedge \mathcal{S}(f_m) = \mathcal{S}(f_y) \wedge \mathit{True} =$ [pdate during 97], which is broader that the mapping a_d obtained with $\mathcal{R}_6$. For this reason, $\mathcal{R}_6$ is a sound rule.

Based on the rule framework, Algorithm *SCM* (in Figure 4) translates simple conjunctions. The algorithm is relatively straightforward. First (in step 1), we evaluate the rules to find the matchings in a given query $\hat{Q}$. As discussed, this matching process effectively partitions $\hat{Q}$ into subsets of inseparable constraints. We then compute the emissions for those subsets as their mappings. The target query is simply the conjunction of all such emissions (step 3).

In addition, we must remove *submatchings* to avoid redundancy (step 2). We can eliminate a matching if it is a subset of some other matching, because the latter will generate a "stricter" mapping with more "underlying constraints." For instance, $\mathcal{R}_6$ defines the mapping to pdate from both the original pyear and pmonth, while $\mathcal{R}_7$ from only the former. Note that $\mathcal{R}_7$ is useful to generate a partial date if pmonth is not constrained in the original query. However, for queries with both pyear and pmonth, such as $\hat{Q}_1$ in Figure 2, $\mathcal{R}_7$ yields a redundant matching $\{f_y\}$, given the larger matching $\{f_y,\ f_m\}$ produced by $\mathcal{R}_6$. We next illustrate Algorithm *SCM* with Example 4. Please refer to [17] for the proof that the algorithm does generate minimal subsuming mappings.

Example 4: Let's translate query $\hat{Q}_1$ in Figure 2 for target *Amazon*. We run Algorithm *SCM* with inputs $\hat{Q}_1$ and K_{Amazon} to show that it does output S_1, *i.e.*, $S_1 = \mathcal{S}(\hat{Q}_1)$.

1. $A \leftarrow \cup[\mathcal{M}(\hat{Q}_1, \mathcal{R}_1), \ldots, \mathcal{M}(\hat{Q}_1, \mathcal{R}_9)] = \cup[\phi, \phi, \{\{f_l\}\}, \{\{f_{t1}\}\}, \phi, \{\{f_y, f_m\}\}, \{\{f_y\}\}, \{\{f_k\}\}, \phi] = \{m_3{:}\{f_l\}, m_4{:}\{f_{t1}\}, m_6{:}\{f_y, f_m\}, m_7{:}\{f_y\}, m_8{:}\{f_k\}\}$
2. We remove the matching m_7 (of $\mathcal{R}_7$), as $m_7 \subset m_6$ (of $\mathcal{R}_6$). Thus, $A = \{m_3, m_4, m_6, m_8\}$.
3. The matchings m_i in A map (by rule $\mathcal{R}_3$, $\mathcal{R}_4$, $\mathcal{R}_6$, $\mathcal{R}_8$) to target queries a_a, a_{t1}, a_d, and $a_{t2} \vee a_{s1}$ (shown on the right top of Figure 2). The output is their conjunction, *i.e.*, $\mathcal{S}(\hat{Q}_1) = a_a \wedge a_{t1} \wedge a_d \wedge (a_{t2}\ \vee\ a_{s1}) = S_1$. ■

Algorithm *SCM*: Simple-Conjunction Mapping
Input: $\hat{Q}$: a simple-conjunction query in the original context; K: mapping rules for a target system T.
Output: $\mathcal{S}(\hat{Q})$: minimal subsuming mapping *w.r.t.* T.
Procedure:
(1) // *find all the matchings for any rule in K:*
- $A \leftarrow \mathcal{M}(\hat{Q},\ K) \equiv \cup[\mathcal{M}(\hat{Q},\ \mathcal{R})]$, for all $\mathcal{R} \in K$

(2) // *remove any matching that is a subset of others:*
- for all $m_i \in A$: for all $m_j \in A$ $(j \neq i)$:
  – if $m_j \subseteq m_i$: remove m_j from A

(3) output $\mathcal{S}(\hat{Q})$ as the conjunction of all the emissions for the remaining matchings in A.

Figure 4: Algorithm *SCM* for mapping simple conjunctions.

Finally, we note that Algorithm *SCM* is quite efficient. To begin with, our rules are very simple– they simply encode the groups of dependent constraints and how they should be mapped. Note that rules are not recursive; matching a rule does not generate new (input) constraints. The matching does not consume constraints either; a constraint can match multiple rules. In other words, rules are independent, and can be evaluated in any order.

We can more formally analyze the running time of Algorithm *SCM* as follows: Given the inputs $\hat{Q}$ and K, let N be the number of constraints in $\hat{Q}$, R the number of rules in K, and P the (maximal) number of constraint patterns in the head of a rule. First, we can perform rule matchings (*i.e.*, step 1 of Algorithm *SCM*) simply by comparing each pattern with each constraint, *i.e.*, the cost will be $N \times P \times R \times m$, where m is a constant. Here we assume independent patterns, *i.e.*, no coupling exists among patterns, such as common variables (*e.g.*, [ln = L] and [fn = L]). We believe this assumption holds in the vast majority of cases. (We actually have no practical counter example.) Next, step 2 compares each pair of matchings; this step can be done in $M^2 \times s$, where M is the number of matchings found in step 1, and s a constant. Finally, in step 3, we fire the rules to generate the mappings for the remaining matchings; the time for this step is $M \times r$, where M is the maximal number of the remaining matchings, and r a constant. Therefore, the worst-case running time is $(N \times P \times R \times m) + (M^2 \times s) + (M \times r)$.

In summary, in the worst-case, the running time is linear in the input size represented by N, P, and R. The quadratic M term is alleviated by the fact that M is in most cases not a large number. In principle, M has an upper bound 2^N, because any subset of the constraints can be a matching. However, M will approach this exponential bound only when there exist extremely *intensive* dependencies such that every subset of the constraints (*e.g.*, on some name and date) cannot be separated in the mapping. Such "high-degree" dependencies are obviously unlikely in practice since we can expect at least some natural schematic conventions

Algorithm *DNF***:** DNF-based Query Mapping
Input: Q: an arbitrary query in the original context; K: mapping rules for a target system T.
Output: $\mathcal{S}(Q)$: minimal subsuming mapping *w.r.t.* T.
Procedure:
(1) convert Q into DNF $\check{Q} = \sum_{i=1}^{m} \hat{D}_i$, where $\hat{D}_i$ is a simple conjunction of constraints.
(2) for each $\hat{D}_i$: $\mathcal{S}(\hat{D}_i) \leftarrow \mathit{SCM}(\hat{D}_i, K)$
(3) return $\mathcal{S}(Q) \equiv \mathcal{S}(\check{Q}) = \sum_{i=1}^{m} \mathcal{S}(\hat{D}_i)$

Figure 5: Algorithm *DNF*.

(*e.g.*, names and dates are typically separated as different attributes). On the other hand, if constraints are all independent, the upper bound will simply be N. We believe that in practice the dependencies will be moderate, and thus the quadratic M term will not be significant.

5 DNF-based Scheme for Complex Queries

In this section we present a first translation algorithm for complex queries with arbitrary Boolean ($\wedge$, $\vee$) combination of constraints. Specific complications arise for such queries because of the implication of the Boolean operators. In particular, can the mapping $\mathcal{S}(\cdot)$ distribute over $\wedge$ and $\vee$? In Example 2 we observed that conjuncts in $Q = C_1 \wedge C_2 = (f_1 \vee f_2) \wedge f_3$ are not separable. In fact, we can handle Q by rewriting its structure, as Example 5 illustrates.

Example 5: Consider Q in Example 2, where the mapping was suboptimal because the separated conjuncts were interrelated. However, if we rewrite Q as $\hat{D}_1$:$(f_1 \wedge f_3) \vee \hat{D}_2$:$(f_2 \wedge f_3)$, it turns out that disjuncts are always separable (according to the results of [15]). Thus, we can handle $\hat{D}_1$ and $\hat{D}_2$ independently, *i.e.*, $\mathcal{S}(Q) = \mathcal{S}(\hat{D}_1) \vee \mathcal{S}(\hat{D}_2)$.

Furthermore, as the disjuncts are simple conjunctions, their mappings can be handled with Algorithm *SCM*. Thus, $\mathcal{S}(Q) = \mathit{SCM}(\hat{D}_1, K_{Amazon}) \vee \mathit{SCM}(\hat{D}_2, K_{Amazon})$. Since the calls to *SCM* fire rule $\mathcal{R}_2$ to handle the matchings $\{f_1, f_3\}$ for $\hat{D}_1$ and $\{f_2, f_3\}$ for $\hat{D}_2$, $\mathcal{S}(Q)$ becomes [author = "Clancy,Tom"] $\vee$ [author = "Klancy,Tom"]. Note that the result is indeed the minimal mapping possible. ■

In general, conjuncts may not be separable, but disjuncts always are. (Reference [15] also studied the general condition, called *inferential completeness*, of when conjuncts are actually separable.) Since disjuncts are always separable, one approach for translation is to first convert all queries into disjunctive normal form (DNF), as was done in Example 5. This approach is followed by Algorithm *DNF* in Figure 5. After the algorithm converts a query, the query has the form $\check{Q} = \sum_{i=1}^{m} \hat{D}_i$, where $\hat{D}_i$ is a simple conjunction. We can distribute the mapping over $\vee$ to each $\hat{D}_i$, because disjuncts are separable, *i.e.*, $\mathcal{S}(\check{Q}) = \sum_{i=1}^{m} \mathcal{S}(\hat{D}_i)$. Furthermore, since each $\hat{D}_i$ is just a simple conjunction, it can be readily handled with Algorithm *SCM*. In fact, Example 5 has illustrated exactly this process.

Unfortunately, although Algorithm *DNF* guarantees the minimal translation, it is expensive, inflexible, and usually unnecessary to rely on DNF. Note that DNF conversion is exponential in the number of constraints (because the Boolean satisfiability problem is NP-complete [22]). To name some problems, first, Algorithm *DNF* requires a *blind* DNF conversion regardless of whether some conjuncts are actually separable. That is, it does not check the potential constraint dependencies to justify the conversion. For instance, if the constraints of Q in Example 5 were on ti instead of ln and fn, the conversion would be unnecessary. (K_{Amazon} shows no inter-dependencies between ti constraints.) Furthermore, the conversion is *global*; it structurally rewrites the whole query. As Section 6 discusses, when conversion is necessary, we can identify and limit its scope to reduce the cost. Lastly, because DNF is typically not a concise Boolean representation, Algorithm *DNF* cannot generate compact translations. (We discuss this compactness in Section 8.) In addition, as we will see in Example 6, Algorithm *DNF* usually requires repeated work (in step 2) to handle the repeating occurrences of the same constraints in many disjuncts. To address these problems, we next discuss a more flexible and efficient scheme that requires local query conversion only when necessary.

6 Traversal-based Top-Down Query Mapping

This section discusses Algorithm *TDQM*, which performs constraint mapping in a top-down traversal of a query tree. Although not essential, for the purpose of explanation, we represent a query in a *query tree*, with interior $\wedge$ and $\vee$ nodes, and leaf constraints. Figure 6 shows a book query $\hat{Q}_{book}$ that we will use as our running example. Recall that $\hat{Q}$ means that the query is conjunctive, while $\check{Q}$ means that it is disjunctive. By viewing $\wedge$ and $\vee$ as n-ary operators that take a set of operands, we generally assume that $\wedge$ and $\vee$ alternate along a path in trees. (Otherwise we can simply collapse any repeating operators, *e.g.*, $\wedge\{a, \wedge\{b, c\}\} = \wedge\{a, b, c\}$.) In other words, the conjuncts in a conjunction $\hat{Q}$ are disjunctive, *i.e.*, $\hat{Q} = \wedge\{\check{C}_i\}$. Similarly, disjuncts are conjunctive, *i.e.*, $\check{Q} = \vee\{\hat{D}_i\}$. Of course, at the leaves, both $\check{C}_i$ and $\hat{D}_i$ can be simply a constraint.

Mapping complex queries is difficult mainly because conjuncts may or may not be separable. Without this complication, translation would be a straightforward top-down traversal of query trees: By distributing $\mathcal{S}(\cdot)$ over $\wedge$ and $\vee$, we eventually would only need to handle leaf constraints (with Algorithm *SCM*). Modulo the conjunction problem, this top-down process is essentially the intuition for Algorithm *TDQM* (Figure 7).

The major challenge in Algorithm *TDQM* is to effectively handle conjunctions, which we will explore in more detail in Section 7. In particular, for inseparable conjuncts, we want to *partition* them into some separable subsets. For instance, as we will see, $\hat{Q}_{book}$ is not separable, *i.e.*, $\mathcal{S}(\hat{Q}_{book}) \neq \mathcal{S}(\check{C}_1)\mathcal{S}(\check{C}_2)\mathcal{S}(\check{C}_3)$. However, it turns out that only $\check{C}_2$ and $\check{C}_3$ are truly dependent; *i.e.*, $\mathcal{S}(\hat{Q}_{book}) = \mathcal{S}(\check{C}_1)\mathcal{S}(\check{C}_2\check{C}_3)$. With the partition of $\{\check{C}_1\}$ and $\{\check{C}_2, \check{C}_3\}$, the mapping can proceed directly to $\check{C}_1$, and we need to rewrite only the subtree $(\check{C}_2 \wedge \check{C}_3)$. We will focus on conjunction separation in Section 7. Here we start with Example 6 to illustrate the top-down traversal approach of Algorithm *TDQM*.

Example 6 (Algorithm *TDQM*): Let us consider mapping $\hat{Q}_{book}$ (Figure 6) for *Amazon* with the rules K_{Amazon} (Figure 3). To begin with, since $\hat{Q}_{book}$ is conjunctive, we must figure out how to separate the conjuncts (or otherwise rewrite the whole query as with Algorithm *DNF*). Section 7.2 will discuss Algorithm *PSafe* specifically for conjunction partition. As we will see, *PSafe*$(\hat{Q}_{book}, K_{Amazon})$ returns two blocks $B_1 = \{\check{C}_1\}$ and $B_2 = \{\check{C}_2, \check{C}_3\}$, *i.e.*, $\mathcal{S}(\hat{Q}_{book}) = \mathcal{S}(\wedge(B_1))\mathcal{S}(\wedge(B_2))$.

We first handle block B_1. As it is a single-conjunct block, the mapping proceeds directly to $\check{C}_1$. Furthermore, since disjuncts are always separable, we can separate $f_l f_f$, f_{k1}, and f_{k2}. Since they are all simple conjunctions (of one or more constraints), we can handle them with Algorithm *SCM*. In summary, by traversing the $\check{C}_1$ subtree, we obtain $\mathcal{S}(\wedge(B_1)) = \mathit{SCM}(f_l f_f, K_{Amazon}) \vee \mathit{SCM}(f_{k1}, K_{Amazon}) \vee \mathit{SCM}(f_{k2}, K_{Amazon})$.

As for B_2, note that intuitively $(\check{C}_2 \wedge \check{C}_3)$ are not separable, because $\check{C}_2$ has a pyear constraint that can combine with either pmonth constraint in $\check{C}_3$ to fire rule $\mathcal{R}_6$ in K_{Amazon}. For inseparable conjuncts, we must rewrite the subtree to continue the mapping. In particular, we can distribute the root $\wedge$ over the next level $\vee$, and thus $\check{B}_2 = f_y f_{m1} \vee f_y f_{m2}$. Intuitively, by pushing down the problematic $\wedge$, we can eventually collect the dependent constraints in some simple conjunctions (*e.g.*, $f_y f_{m1}$ and $f_y f_{m2}$). As we rewrite $\wedge(B_2)$ to a disjunctive form, the mapping can proceed to the new disjuncts, *i.e.*, $\mathcal{S}(\wedge(B_2)) = \mathit{SCM}(f_y f_{m1}, K_{Amazon}) \vee \mathit{SCM}(f_y f_{m2}, K_{Amazon})$. Thus, the complete mapping of $\hat{Q}_{book}$ is $\mathcal{S}(\wedge(B_1))\mathcal{S}(\wedge(B_2))$.

Observe that during tree traversal, our algorithm actually rewrites the query. In particular, $\hat{Q}_{book}$ is effectively converted to $(f_l f_f \vee f_{k1} \vee f_{k2}) \wedge (f_y f_{m1} \vee f_y f_{m2})$ so that dependent constraints are collected in simple conjunctions. Note that, in comparison, Algorithm *DNF* would require a global and blind conversion into DNF: $(f_l f_f f_y f_{m1} \vee f_l f_f f_y f_{m2} \vee f_{k1} f_y f_{m1} \vee f_{k1} f_y f_{m2} \vee f_{k2} f_y f_{m1} \vee f_{k2} f_y f_{m2})$. Furthermore, mapping based on DNF requires more work because it is typically not as concise as the original tree. Therefore, in different invocations of Algorithm *SCM* we need to repeatedly handle those repeating constraints in various disjuncts (*e.g.*, f_y appears in all the disjuncts of the above DNF, and f_{m1} in three of them). ■

As Example 6 informally illustrated, Algorithm *TDQM* traverses a given query tree to perform the mapping. We structure this tree traversal as a recursive procedure in Figure 7. The procedure differentiates three cases: At an $\vee$-node (Case-1), it simply separates and recursively calls *TDQM* on each disjunct. For complex conjunctions (Case-2), it calls upon Algorithm *PSafe* to determine the partition of conjuncts, and handle each block independently.

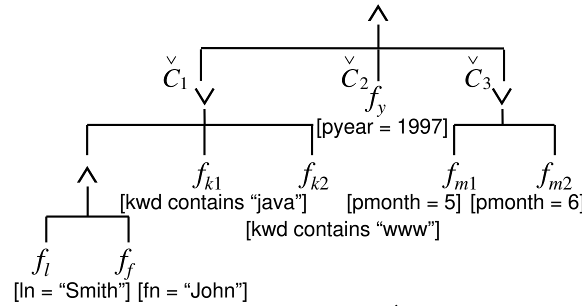

Figure 6: A query tree $\hat{Q}_{book}$.

Algorithm *TDQM*: Top-Down Query Mapping
Input: Q: an arbitrary query in the original context; K: mapping rules for a target system T.
Output: $\mathcal{S}(Q)$: minimal subsuming mapping $w.r.t.$ T.
Procedure:
//**Case-1: disjunctive**; *$\vee$-node.*
• if $Q = \vee(\{\hat{D}_1, \hat{D}_2, \ldots, \hat{D}_n\})$:
 – for each $\hat{D}_i$: $\mathcal{S}(\hat{D}_i) \leftarrow$ *TDQM*$(\hat{D}_i, K)$ //*recursive call.*
 – return $\mathcal{S}(Q) \leftarrow \vee(\{\mathcal{S}(\hat{D}_1), \ldots, \mathcal{S}(\hat{D}_n)\})$ //*separate* $\hat{D}_i$.
//**Case-2: conjunctive**; *$\wedge$-node with some non-leaf children.*
• else if $Q = \wedge(\{\check{C}_1, \check{C}_2, \ldots, \check{C}_n\})$ with some non-leaf $\check{C}_i$:
 – $P \leftarrow$ *PSafe*(Q, K) //*partition Q into separable blocks.*
 – for each $B \in P$:
  – $\check{B} \leftarrow$ *Disjunctivize*$(\wedge(B))$ //*local query rewriting.*
  – $\mathcal{S}(\wedge(B)) \leftarrow$ *TDQM*$(\check{B}, K)$ //*recursive call.*
 – return $\mathcal{S}(Q) \leftarrow \wedge_{B \in P}\mathcal{S}(\wedge(B))$
//**Case-3: simple conjunctions**; *leaf or $\wedge$ of some leaves.*
• else if Q is a simple conjunction:
 – return $\mathcal{S}(Q) \leftarrow$ *SCM*(Q, K) //*call Algorithm SCM.*

Function *Disjunctivize*$(\wedge(B))$://*rewrite to a disjunctive form.*
//*suppose* $B = \{\check{C}_1, \ldots, \check{C}_k\}$*, and* $\check{C}_i = \vee(\{\hat{D}_{i1}, \ldots, \hat{D}_{im_i}\})$.
• if $k = 1$: return $\check{B} \leftarrow \check{C}_1$ //*single-conjunct block.*
• else: //*distribute $\wedge$ over $\vee$, e.g.,* $\wedge\{(\hat{D}_{11} \vee \hat{D}_{12}), (\hat{D}_{21} \vee \hat{D}_{22})\}$ //*becomes* $\vee\{\hat{D}_{11}\hat{D}_{21}, \hat{D}_{11}\hat{D}_{22}, \hat{D}_{12}\hat{D}_{21}, \hat{D}_{12}\hat{D}_{22}\}$
 – return $\check{B} \leftarrow \vee(\{\hat{D}_{1j_1}\hat{D}_{2j_2} \ldots \hat{D}_{kj_k} \mid j_i \in [1 : m_i]\})$

Figure 7: Algorithm *TDQM* for mapping arbitrary queries.

Eventually, at (the conjunction of) leaves (Case-3), it relies on Algorithm *SCM* to process simple conjunctions, which is actually the base case that terminates the recursion.

In particular, at a conjunction, we rewrite *locally* and *incrementally* each inseparable block into a disjunctive form. As Figure 7 (bottom) shows, function *Disjunctivize* converts a conjunctive subtree by distributing the $\wedge$ at the root over the $\vee$ at the next level. For instance, in Example 6 we rewrote $(f_y) \wedge (f_{m1} \vee f_{m2})$ to $(f_y f_{m1} \vee f_y f_{m2})$; the rewriting was localized to block $\{\check{C}_2, \check{C}_3\}$. Furthermore, Algorithm *TDQM* performs such rewritings incrementally instead of directly into DNF. For instance, suppose A, B, and C are complex queries. After *Disjunctivize* converts $(A \vee B)(C)$ into $(AC \vee BC)$, if the dependency is between A and C, we need not to further rewrite BC at all.

We have presented Algorithm *TDQM*, which maps constraints in the top-down traversal of a query tree and performs structure conversion only when necessary. Therefore, the remaining challenge is the partition of conjuncts that respects constraint dependencies. We study this problem next.

7 Conjunct Separation

This section discusses how we can separate a conjunction. First, as a basis, Section 7.1 studies the *safety* conditions for conjunct separation (*i.e.*, when it is safe to translate conjuncts independently). Section 7.2 then informally sketches Algorithm *PSafe*, which actually partitions conjuncts safely.

7.1 Safety Conditions for Conjunct Separability

We now explore how to determine if a conjunction $\hat{Q} = C_1 \cdots C_n$ can be separated safely (*i.e.*, without impacting constraint mapping). We first study the base case when the conjunct C_i's are simple conjunctions, and then the general case when C_i's are disjunctive. Note that, while the former is not a "natural" pattern in our query trees (that assume alternating $\wedge$ and $\vee$), it is the basis for the general case.

Base Case: Simple-Conjunction Conjunctions

We first focus on $\hat{Q} = \hat{C}_1 \cdots \hat{C}_n$ when $\hat{C}_i$'s are simple conjunctions, to determine the safety condition that ensures $\mathcal{S}(\hat{Q}) = \mathcal{S}(\hat{C}_1) \cdots \mathcal{S}(\hat{C}_n)$. Note that since $\hat{C}_i$'s as well as the entire $\hat{Q}$ are all simple conjunctions, their mappings can be handled with Algorithm *SCM*. Thus, for some mapping rules K, the separability is to see if $\mathit{SCM}(\hat{Q}, K) = \mathit{SCM}(\hat{C}_1, K) \cdots \mathit{SCM}(\hat{C}_n, K)$. As Algorithm *SCM* is essentially a rule matching process, if all the matchings in $\mathit{SCM}(\hat{Q}, K)$ can also be found in some $\mathit{SCM}(\hat{C}_i, K)$, then the condition must hold true. In other words, $\hat{Q}$ is separable when no matchings occur across the conjuncts. Example 7 illustrates this intuition, and then Definition 2 formally states when conjunction $\hat{Q}$ is safe to be separated.

Example 7: Let $\hat{Q} = \hat{C}_1{:}(f_l f_f) \wedge \hat{C}_2{:}(f_y) \wedge \hat{C}_3{:}(f_{m1})$ (part of the query in Figure 6). For rules K_{Amazon} (Figure 3) representing target *Amazon*, $\hat{Q}$ is not separable because of the matching $\{f_y, f_{m1}\}$ (for rule $\mathcal{R}_6$), which can only be found when we consider $\hat{Q}$ as a whole. That is, m is a *cross-matching* that appears in $\mathcal{M}(\hat{Q}, K_{Amazon})$ (*i.e.*, the matchings from $\hat{Q}$ for any rule in K_{Amazon}) but not in any $\mathcal{M}(\hat{C}_i, K_{Amazon})$. Those conjuncts that contain a cross-matching (in this case $\hat{C}_2$ and $\hat{C}_3$) cannot be separated, or else the cross-matching will be adversely omitted.

In particular, if we separate each $\hat{C}_i$, the mapping will miss the target constraint [pdate during May/97] (generated by $\mathcal{R}_6$ from matching $\{f_y, f_{m1}\}$). In fact, it will drop the month component, because with the separation $\mathcal{R}_7$ will fire instead (with matching $\{f_y\}$ from $\hat{C}_2$). ■

Definition 2 (Safety for Base-Case Conjunctions): Let $\hat{Q}$ = $\hat{C}_1 \cdots \hat{C}_n$, where $\hat{C}_i$'s are simple conjunctions. $\hat{Q}$ is *safe w.r.t.* rules K if $\mathcal{M}(\hat{Q}, K) - \cup_{i=1}^{n} \mathcal{M}(\hat{C}_i, K) = \phi$; otherwise $\hat{Q}$ is *unsafe*. ■

Note that safety is sufficient but not necessary for separability. Namely, a cross-matching might be "redundant," and thus its omission by conjunct separation has no impact on the mapping. While this redundancy is rare in practice, to illustrate we show a somehow artificial example in [17]. Furthermore, to complete our discussion, reference [17] actually presents the precise (*i.e.*, sufficient and necessary) condition for conjunct separation. However, we note there that it can be expensive to fully test the precise condition. In fact, we believe that in practice a cross-matching is unlikely to be redundant, and the test of Definition 2 will be adequate. If we use Definition 2 and encounter a rare redundant cross-matching, we will have to pay the cost of an extra query conversion, but the mapping will still be minimal.

General Case: Disjunctive-Query Conjunctions

Conjunctions in our query trees generally have the form $\hat{Q} = \check{C}_1 \cdots \check{C}_n$, where $\check{C}_i$'s are disjunctive with "ingredient" disjuncts I_{ij}, *i.e.*, $\check{C}_i = I_{i1} \vee \cdots \vee I_{im_i}$. (The ingredients I_{ij} can themselves be complex queries.) Since $\check{C}_i$'s are conjuncts, any combinations of their ingredients of the form $\hat{D} = I_{1k_1} \cdots I_{nk_n}$ is an implicit conjunction in $\hat{Q}$. Intuitively, $\hat{Q}$ is separable when there is no interdependencies among the ingredients from different $\check{C}_i$'s. In other words, when all such "ingredient conjunctions" are separable, *i.e.*, $\mathcal{S}(\hat{D}) = \mathcal{S}(I_{1k_1}) \cdots \mathcal{S}(I_{nk_n})$, then $\hat{Q}$ as the "whole conjunction" must also be separable, which we illustrate with Example 8.

Example 8: Suppose $\hat{Q} = \check{C}_1 \check{C}_2 = (I_{11} \vee I_{12})(I_{21})$. (We can view $\check{C}_2$ as disjunctive with one disjunct.) To see the ingredient conjunctions, let's convert $\hat{Q}$ into a disjunctive form with function *Disjunctivize* (Figure 7). That is, we compute $\check{Q} = \mathit{Disjunctivize}(\hat{Q}) = \vee(\{\hat{D}_1{:}I_{11}I_{21}, \hat{D}_2{:}I_{12}I_{21}\})$.

We want to show that if the ingredient conjunctions ($\hat{D}_1$ and $\hat{D}_2$) are separable, then so is $\hat{Q}$, *i.e.*, $\mathcal{S}(\hat{Q}) = \mathcal{S}(\check{C}_1)\mathcal{S}(\check{C}_2)$. Since $\hat{Q} = \hat{D}_1 \vee \hat{D}_2$, the left hand side $\mathcal{S}(\hat{Q})$ is equivalent to $\mathcal{S}(\hat{D}_1) \vee \mathcal{S}(\hat{D}_2)$ (disjuncts are always separable), or $\mathcal{S}(I_{11})\mathcal{S}(I_{21}) \vee \mathcal{S}(I_{12})\mathcal{S}(I_{21})$ because $\hat{D}_1$ and $\hat{D}_2$ are separable. The right hand side $\mathcal{S}(\check{C}_1)\mathcal{S}(\check{C}_2)$ is also equivalent to the last expression, since $\mathcal{S}(\check{C}_1) = \mathcal{S}(I_{11}) \vee \mathcal{S}(I_{12})$ and $\mathcal{S}(\check{C}_2) = \mathcal{S}(I_{21})$. ■

Example 8 suggests the following safety condition. Note that Definition 3 defines safety recursively; as we will see, Definition 2 is the base case that grounds the recursion.

Definition 3 (Safety for General-Case Conjunctions): Let $\hat{Q} = \check{C}_1 \cdots \check{C}_n$, where $\check{C}_i$'s are disjunctive, *i.e.*, $\check{C}_i = I_{i1} \vee \cdots \vee I_{im_i}$, and I_{ij}'s are arbitrary queries. Let $\check{Q} = \mathit{Disjunctivize}(\hat{Q})$. $\hat{Q}$ is *safe w.r.t.* rules K if all the disjuncts (as a conjunction $I_{1k_1} \cdots I_{nk_n}$) in $\check{Q}$ are safe (and thus separable) *w.r.t.* K; otherwise, $\hat{Q}$ is *unsafe*. ■

Note that, while we can separate a safe conjunction, an unsafe one might actually be separable. To illustrate these rare cases, consider $\hat{Q} = \check{C}_1 \check{C}_2 = (x \vee y)(z)$. Suppose that $\{y, z\}$ (among others) is a matching for the mapping rules. Note that $\hat{Q}$ is unsafe, because the combination $(y)(z)$ is unsafe (since $\{y, z\}$ is a cross-matching). Thus $\hat{Q}$ will normally be inseparable. However, in the particular case when there is no mapping for either $\{x\}$ or $\{x, z\}$ (and thus $\mathcal{S}(x) = True$ and $\mathcal{S}(xz) = \mathcal{S}(z)$), we can show that $\mathcal{S}(\hat{Q}) = \mathcal{S}(\check{C}_1)\mathcal{S}(\check{C}_2)$: First, $\mathcal{S}(\hat{Q}) = \mathcal{S}(xz \vee yz) = \mathcal{S}(xz) \vee \mathcal{S}(yz) = \mathcal{S}(z) \vee \mathcal{S}(yz)$. Thus we obtain $\mathcal{S}(\hat{Q}) = \mathcal{S}(z)$, since $\mathcal{S}(z) \supseteq \mathcal{S}(yz)$. Now consider the mapping of the other way: $\mathcal{S}(\check{C}_1)\mathcal{S}(\check{C}_2) = \mathcal{S}(x \vee y)\mathcal{S}(z) = [\mathcal{S}(x) \vee \mathcal{S}(y)]\mathcal{S}(z)$. Thus $\mathcal{S}(\check{C}_1)\mathcal{S}(\check{C}_2)$ also simplifies to $\mathcal{S}(z)$ since $\mathcal{S}(x) = True$. Therefore, $\hat{Q}$ is actually separable while being unsafe. Observe that this "anomaly" is solely because (the mapping of) the unsafe term $(y)(z)$ is "masked" by $\mathcal{S}(xz) = \mathcal{S}(z)$, which would not occur if $\mathcal{S}(x) \neq True$.

To explain the anomalies, we also present the precise separability condition for the general-case conjunctions in [17]. However, such anomalies should be rare in practice. That is, we believe that we can use Definition 3, and very seldom misdiagnose a conjunction as not separable. Again, the misdiagnosis simply means that the resulting mapping may not be the most succinct, but it will still be minimal.

Testing the Safety Conditions

We next discuss how to efficiently test the safety conditions (to determine separability). In principle, to check if $\hat{Q} = \check{C}_1 \cdots \check{C}_n$ is safe, we can recursively apply Definition 3. As each application will "*Disjunctivize*" the query, eventually we will deal with the base case (when all the $\check{C}_i$'s become simple conjunctions) in Definition 2.

In fact, we can first convert $\check{C}_i$'s into DNF to avoid the recursion: Note that, in Definition 3, when all $\check{C}_i$'s are in DNF, the ingredients I_{ij} are just simple conjunctions. Therefore, we can check the safety of $I_{1k_1} \cdots I_{nk_n}$ with Definition 2. To illustrate, consider (in Figure 6) $\hat{Q}_{book} = \check{C}_1\check{C}_2\check{C}_3 = (f_l f_f \vee f_{k1} \vee f_{k2})(f_y)(f_{m1} \vee f_{m2})$. Since $\check{C}_i$'s are already in DNF, we then check the safety for all the six conjunctions $\hat{D}_1$: $(f_l f_f)(f_y)(f_{m1})$, $\hat{D}_2$: $(f_l f_f)(f_y)(f_{m2})$, *etc*. Applying Definition 2, we conclude that (among others) $\hat{D}_1$ is unsafe with the cross-matching (for rules K_{Amazon}) $\{f_y, f_{m1}\}$. Thus, $\hat{Q}_{book}$ is unsafe.

However, this "brute-force" approach is not as efficient as possible; it unnecessarily relies on $\check{C}_i$'s *full* DNF. (As discussed, DNF can be expensive to compute, and it contains more terms to check.) The key intuition for making this process more efficient is that the safety conditions ultimately depend solely on the existence of cross-matchings. Therefore, we can omit from $\check{C}_i$'s those constraints that will not contribute to forming a cross-matching, and thus focus on those *may*. We call the DNF of such a simplified $\check{C}_i$ the *essential DNF* (or *E*DNF), and write it as $\mathcal{D}_e(\check{C}_i)$. While omitting "useless" terms from $\check{C}_i$ does not impact the safety results, in most cases it will greatly simplify the safety check. We illustrate by redoing the $\hat{Q}_{book}$ example.

Example 9 (Essential DNF): Consider again (in Figure 6) $\hat{Q}_{book} = \check{C}_1\check{C}_2\check{C}_3 = (f_l f_f \vee f_{k1} \vee f_{k2})(f_y)(f_{m1} \vee f_{m2})$. The *E*DNF's are $\mathcal{D}_e(\check{C}_1) = \epsilon$, $\mathcal{D}_e(\check{C}_2) = f_y$, and $\mathcal{D}_e(\check{C}_3) = f_{m1} \vee f_{m2}$. Intuitively, only those "essential" constraints (*i.e.*, f_y, f_{m1}, and f_{m2}) involved in the potential cross-matchings (namely $\{f_y, f_{m1}\}$ and $\{f_y, f_{m2}\}$) remain in the *E*DNF. Note that we use ϵ to represent "something unimportant" (for testing safety) or "*don't care.*"

Replacing each $\check{C}_i$ by $\mathcal{D}_e(\check{C}_i)$, we can then check the safety with the simplified expression $(\epsilon)(f_y)(f_{m1} \vee f_{m2})$. In turn, we will check the safety for simple conjunctions $\hat{D}'_1 = (\epsilon)(f_y)(f_{m1})$ and $\hat{D}'_2 = (\epsilon)(f_y)(f_{m2})$. Obviously, testing the safety for these $\hat{D}'_i$'s involves less work than that for $\hat{D}_i$'s (based on the full DNF) just illustrated, because using $\check{C}_i$'s *E*DNF results in fewer and simpler terms. Note that we indeed obtain the same result that $\hat{Q}_{book}$ is unsafe, since all the cross-matchings (*i.e.*, $\{f_y, f_{m1}\}$ and $\{f_y, f_{m2}\}$) are preserved through the simplification. ■

Given a query tree, we use Procedure *EDNF* to compute the *E*DNF for every subquery in a bottom-up tree traversal. We present the details of this procedure in [17]. Here we only stress that using *E*DNF allows us to focus on only the essential terms that may potentially contribute to cross-matchings. In particular, when a query does not contain any constraint dependencies (in which case there are no multi-constraint matchings), then all the *E*DNF will simply be ϵ. With this reduction, the safety check has virtually no cost.

7.2 Partitioning Conjunctive Queries

Based on the safety conditions, we next study how to safely separate conjuncts. This section sketches Algorithm *PSafe*, the critical technique that Algorithm *TDQM* (Figure 7) relies on for partitioning conjunctions. Due to space limitations, we will simply illustrate the ideas and leave the full details of Algorithm *PSafe* for [17].

Specifically, when a conjunction $\check{C}_1 \cdots \check{C}_n$ is safe, our algorithm simply returns the n blocks $\{\check{C}_1\}, \ldots, \{\check{C}_n\}$, which means that every conjunct can be independently translated. Otherwise, for an unsafe conjunction, Algorithm *PSafe* can collect those inseparable conjuncts in the same block. This partition can limit the query structure conversion to within a block. Note that we can instead simply convert the whole unsafe conjunction into a disjunction (or even directly into DNF as in Algorithm *DNF*). However, such blind conversion is not necessary since not all the conjuncts in an unsafe conjunction are interrelated.

More formally, for a conjunction $\hat{Q} = \check{C}_1 \cdots \check{C}_n$, a *partition* P is a set of *blocks* B_j, *i.e.*, $P = \{B_1, \ldots, B_m\}$. Each block contains some conjuncts $\check{C}_i$. For instance, for query $\hat{Q}_{book}$ (Example 9) the partition will have two blocks $B_1 = \{\check{C}_1\}$ and $B_2 = \{\check{C}_2, \check{C}_3\}$. We require that each conjunct $\check{C}_i$ be handled in exactly one block (so that $\check{C}_i$ does not repeat in the mapping). Note that the original conjunction can be written as $\hat{Q} = \hat{B}_1 \cdots \hat{B}_m$, where $\hat{B}_j$ is the conjunction of block B_j (*i.e.*, $\hat{B}_j = \wedge(B_j)$). For query mapping the partition must be *safe*, *i.e.*, $\mathcal{S}(\hat{Q}) = \mathcal{S}(\hat{B}_1) \cdots \mathcal{S}(\hat{B}_m)$. In our example, we can verify that $\mathcal{S}(\hat{Q}) = \mathcal{S}(\hat{B}_1)\mathcal{S}(\hat{B}_2) = \mathcal{S}(\check{C}_1)\mathcal{S}(\check{C}_2\,\check{C}_3)$; the problematic matchings $\{f_y, f_{m1}\}$ and $\{f_y, f_{m2}\}$ are both contained in

$\check{C}_2\check{C}_3$. In addition, we want the blocks to be *minimal*, *i.e.*, no B_j can be further safely partitioned into smaller blocks. In our $\hat{Q}_{book}$ example, we cannot separate block $\{\check{C}_2, \check{C}_3\}$.

The partition algorithm extends our discussion for testing the safety conditions (Section 7.1). Recall that we compute the *E*DNF of conjuncts (with Algorithm *EDNF* detailed in [17]), and check if any cross-matchings exist across the combinations of the *E*DNF ingredients, as Example 9 illustrated. Based on this same approach, our partition algorithm further finds the blocks of conjuncts that *cover* (or contain) the identified matchings. By covering all the cross-matching, we ensure that the resulting blocks are safe to separate. Example 10 illustrates this extension.

Example 10: We continue Example 9 to partition conjunction $\hat{Q}_{book}$. In Example 9, we found two cross-matchings: $m_1 = \{f_y, f_{m1}\}$ and $m_2 = \{f_y, f_{m2}\}$. To partition $\hat{Q}_{book}$, we then find the blocks that cover the matchings: Since m_1 is a matching contributed by $\check{C}_2$ and $\check{C}_3$, we consider $B = \{\check{C}_2, \check{C}_3\}$ as a (candidate) block for the partition. Similarly, m_2 is also covered by the same block. Since m_1 and m_2 are both exclusively covered by block B, the partition must include B to cover either matching. Finally, because $\check{C}_1$ does not participate in any cross-matchings, it is a block by itself. Therefore, the partition is $\{\{\check{C}_1\}, \{\check{C}_2, \check{C}_3\}\}$. ■

Essentially, as Example 10 illustrated, our partition algorithm will find the blocks that are *necessary* to cover all the cross-matchings. On the other hand, not all the blocks that cover some cross-matchings are required in the partition. Otherwise (if we include all such candidate blocks) the partition might not be minimal, which means some blocks can be further separated. We next illustrate the idea. (Note that, to simplify presentation, in Example 11 we do not actually compute the conjunct *E*DNF.)

Example 11: Consider $\hat{Q}_a = \check{C}_1\check{C}_2\check{C}_3 = (x)(y)(yu \vee v)$. Assume that the matchings for constraints x, y, u, v are $\{x, y\}$, $\{u\}$, and $\{v\}$. Apparently, the partition needs blocks $\{\check{C}_1, \check{C}_2\}$ and $\{\check{C}_1, \check{C}_3\}$ as they both cover the matching $\{x, y\}$. This partition (that includes both blocks) is not minimal: It turns out that only $\{\check{C}_1, \check{C}_2\}$ is necessary, *i.e.*, $\mathcal{S}(\hat{Q}_a) = \mathcal{S}(\check{C}_1\check{C}_2)\mathcal{S}(\check{C}_3)$. In fact, we can verify that $\hat{Q}_a \equiv (x)(y)(u \vee v)$, and thus clearly we can separate $\check{C}_1$ and $\check{C}_3$.

To contrast, for the same constraints, consider another query $\hat{Q}_b = \check{C}_1\check{C}_2\check{C}_3 = (x)(y \vee u)(y \vee v)$. Again, the matching $\{x, y\}$ appears across $\check{C}_1$ and $\check{C}_2$ as well as $\check{C}_1$ and $\check{C}_3$. However, unlike the previous case, now both blocks $\{\check{C}_1, \check{C}_2\}$ and $\{\check{C}_1, \check{C}_3\}$ are required. Consequently we will merge the overlapping blocks (so that $\check{C}_1$ will not be handled twice). Thus the partition is the single block $\{\check{C}_1, \check{C}_2, \check{C}_3\}$, *i.e.*, we will evaluate $\mathcal{S}(\hat{Q}_b)$ as $\mathcal{S}(\check{C}_1\check{C}_2\check{C}_3)$. ■

We have illustrated the essential idea for conjunction partitioning. In [17] we present the details of Algorithm *PSafe* (and illustrate how we actually partition the queries in Example 11). In summary, the algorithm first finds all the "candidate" blocks that cover some cross-matchings, and then selects a minimal set of blocks to ensure that all the cross-matchings are covered (which is indeed a minimal cover problem). Note that this partitioning technique is essential to enable the effective handling of conjunctions. As Section 6 discussed, our translation mechanism (Algorithm *TDQM*) relies on Algorithm *PSafe* to determine the separation of conjuncts, and thus avoid the blind DNF conversions otherwise. Our discussion of Algorithm *PSafe* completes the overall translation framework.

8 Optimality, Compactness, and Complexity

Our algorithms produce the best mapping possible, *i.e.*, the translated queries are the most selective while still subsuming the original ones. This guarantee comes from two facts: First, our basic rules codify the human expertise that directs the best mapping for individual groups of dependent constraints. Second, our algorithms correctly handle conjunctions; in particular, the separation of conjuncts respects constraint dependencies. It is intuitive to see that Algorithm *TDQM* does produce minimal subsuming mappings: In the top-down traversal of a query tree, *TDQM* distributes the mapping over $\vee$ because disjuncts are always separable. For conjunctions, our algorithms separate only those conjuncts that meet the safety conditions (Definition 2 and 3). The safety conditions ensure that no dependencies exist among separated conjuncts. Lastly, we handle the base case with Algorithm *SCM*, which we know guarantees the correctness for simple conjunctions. We give the formal proof for this optimality as well as the safety conditions in [17].

Furthermore, Algorithm *TDQM* generates more compact mappings (with fewer terms) as compared to the DNF-based algorithm (as Example 6 illustrated). Note that, although term minimization [23] is possible, DNF is inherently not a compact representation for Boolean functions as restricted by the two-level structure. In contrast, Algorithm *TDQM* does not use DNF; it calls upon Algorithm *PSafe* to collect conjuncts (for structure rewriting) to meet the safety conditions. Unless the safety conditions give a *false negative* (which we believe to be rare), our algorithms will rewrite a subquery only if necessary. To quantify, let's measure the *compactness* of a query as the number of nodes in the parse tree. For a Boolean expression with n constraints, the least compact DNF (*i.e.*, the *canonical* DNF) can have up to 2^n minterms, and each minterm is a conjunction of n constraints. Thus the compactness is on the order of $2^n \times n$. In contrast, the most compact tree for such an expression would be on the order of n nodes (*i.e.*, the number of constraints). Because our algorithm preserves the query structure whenever possible, the *worst-case* compactness ratio can be as large as $(2^n \times n)/n$, *i.e.*, 2^n. That is, there may be cases where our scheme will yield a query that is 2^n times *smaller* than a query produced via DNF conversion. Obviously, this ratio can be arbitrarily large for large queries.

We also note that, while carefully addressing constraint dependencies, our algorithm is quite efficient. In fact, when a query does not involve dependent constraints, our algorithm pays virtually no extra cost (in addition to the mapping of single constraints). Recall that we address dependencies among conjuncts by checking the safety conditions. As Section 7.1 discussed, we can check the safety for $\hat{Q} = \check{C}_1 \cdots \check{C}_n$ brute force by first converting each $\check{C}_i$ as well as $\hat{Q}$ to their full DNF's (instead of using *E*DNF). We then check through all the disjuncts in the DNF of $\hat{Q}$. In the worst case, $\hat{Q}$ can have up to 2^{nk} disjuncts, where n is the number of conjuncts $\check{C}_i$ and k the (maximal) number of constraints in each $\check{C}_i$. Thus this brute-force approach has a "blind" cost on the order of 2^{nk}.

In contrast, our approach based on *E*DNF will pay a cost "proportional" to the degree of dependency (informally speaking). Recall that we use *E*DNF that eliminates useless terms (Example 9). In other words, $\check{C}_i$'s *E*DNF will only contain those constraints that participate in potential matchings spanning beyond $\check{C}_i$. If e is the number of those "essential" constraints remaining, the *E*DNF of $\check{C}_i$ will have an upper bound of 2^e terms. Multiplying all such terms from different $\check{C}_i$'s, we obtain a total of 2^{ne} disjuncts to check. Therefore, this cost (on the order 2^{ne}) is actually a function of the degree of dependency as represented by e. For instance, when there is no dependency, we have $e = 0$, *i.e.*, the *E*DNF's of $\check{C}_i$'s are simply ϵ (*e.g.*, $\mathcal{D}_e(\check{C}_1) = \epsilon$ in Example 9). Therefore, we only need to check one term (*i.e.*, $2^{ne} = 1$) consisting of all ϵ; thus there is virtually no cost. In contrast, the DNF approach still pays the cost of 2^{nk}, which can be arbitrarily large depending on the query size.

9 Conclusion

In this paper we presented a framework as well as the associated algorithms for translating constraint queries across heterogeneous information systems. As we discussed, our algorithms produce query mappings that are both optimal and the most compact possible. Furthermore, our algorithms are efficient; Algorithm *SCM* runs in time linear to the input size, and Algorithm *TDQM* pays virtually no extra cost when no constraint dependencies exist.

We have implemented a running prototype for query mapping in the Stanford Digital Libraries Project. This prototype was based on our earlier work [15, 21] that did not address potential constraint dependencies and did not provide a mapping rule system. The deficiencies of this implementation motivated the work described in this paper. We are in the process of extending the prototype with the algorithms discussed in this paper.

References

[1] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):51–60, 1992.

[2] J. D. Ullman. Information integration using logical views. In *Proc. ICDT*, Delphi, Greece, Jan. 1997. Springer, Berlin.

[3] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. VLDB*, Bombay, India, 1996.

[4] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Query-answering algorithms for information agents. In *Proc. of the 13th National Conf. on Artificial Intelligence, AAAI-96*, Portland, Oreg., Aug. 1996. AAAI Press, Menlo Park, Calif.

[5] Y. Papakonstantinou, H. García-Molina, and J. Ullman. Medmaker: A mediation system based on declarative specifications. In *Proc. ICDE*, New Orleans, La., 1996.

[6] Y. Papakonstantinou, H. García-Molina, A. Gupta, and J. Ullman. A query translation scheme for rapid implementation of wrappers. In *Proc. of the 4th International Conf. on Deductive and Object-Oriented Databases*, Singapore, Dec. 1995.

[7] O. M. Duschka. *Query Planning and Optimization in Information Integration*. PhD thesis, Stanford Univ., 1997.

[8] M. R. Genesereth, A. M. Keller, and O. M. Duschka. Infomaster: An information integration system. In *Proc. of the 1997 ACM SIGMOD Conf.*, Tucson, Ariz., 1997.

[9] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proc. VLDB*, pages 276–285, Athens, Greece, Aug. 1997.

[10] M. T. Roth and P. M. Schwarz. Don't scrap it, wrap it! a wrapper architecture for legacy data sources. In *Proc. VLDB*, pages 266–275, Athens, Greece, Aug. 1997.

[11] O. Kapitskaia, A. Tomasic, and P. Valduriez. Dealing with discrepancies in wrapper functionality. Technical Report RR-3138, INRIA, 1997.

[12] H. García-Molina, W. Labio, and R. Yerneni. Capability sensitive query processing on internet sources. In *Proc. ICDE*, Sydney, Australia, Mar. 1999.

[13] A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering queries using templates with binding patterns. In *Proc. PODS*, San Jose, Calif., May 1995.

[14] A. Y. Levy, A. Rajaraman, and J. D. Ullman. Answering queries using limited external query processors. In *Proc. PODS*, Montreal, Canada, June 1996.

[15] C.-C. K. Chang, H. García-Molina, and A. Paepcke. Boolean query mapping across heterogeneous information sources. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):515–521, Aug. 1996.

[16] C.-C. K. Chang, H. García-Molina, and A. Paepcke. Boolean query mapping across heterogeneous information sources (extended version). Technical Report SIDL-WP-1996-0044, Stanford Univ., Sept. 1996. `http://www-diglib.-stanford.edu`.

[17] C.-C. K. Chang and H. García-Molina. Mind your vocabulary: Query mapping across heterogeneous information sources (extended version). Technical Report SIDL-WP-1998-0095, Stanford Univ., 1999. `http://www-diglib.-stanford.edu`.

[18] S. Heiler. Semantic interoperability. *Computing Surveys*, 27(2):271–273, June 1995.

[19] Y. Papakonstantinou, A. Gupta, and L. Haas. Capabilities-based query rewriting in mediator systems. In *Proc. PDIS*, Miami Beach, Flor., 1996.

[20] C.-C. K. Chang and H. García-Molina. Conjunctive constraint mapping for data translation. In *Proc. of the Third ACM International Conf. on Digital Libraries*, Pittsburgh, Pa., June 1998. ACM Press, New York.

[21] C.-C. K. Chang, H. García-Molina, and A. Paepcke. Predicate rewriting for translating boolean queries in a heterogeneous information system. *ACM Transactions on Information Systems*, 17(1):1–39, Jan. 1999.

[22] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

[23] E. J. McCluskey. *Logic Design Principles*. Prentice Hall, Englewood Cliffs, N.J., 1986.