Mind Your Vocabulary: Query Mapping Across Heterogeneous Information Sources *
(Extended Version)

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.

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.*

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

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.*, Information Manifold [3, 4], TSIMMIS [5, 6], Infomaster [7, 8], Garlic [9, 10], DISCO [11], and others [12, 13, 14]. We discuss this 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).

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. Our discussion assumes a simple relational view of data. Specifically, wrappers present each source as a set of *source relations*. We believe our framework is not sensitive to the data models; *e.g.*, in reference [17] we discuss the translation of hierarchical data.

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. For simplicity, we may write a selection constraint as [attr1 op value] when the containing view of attr1 is clear from the context (such as in Example 1 and Example 2 where we considered only one integrated view).

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. Note that in general a view can be a union of SPJ components; *e.g.*, a *book* view can be a union of two relations from two bookstore sources. In this case, we can process each component separately and union the results as typically done.

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.

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

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.*,

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

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

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

We next illustrate this translation problem with Example 3, which considers a mediator that integrates two sources.

Example 3: To illustrate the translation problem, let us consider a mediator for two sources. Suppose that source T_1 provides relation *paper*(ti, au) for paper titles and authors and *aubib*(name, bib) for author names and their bibliography. Source T_2 has *prof*(ln, fn, dept) for professor last, first names, and departments. The

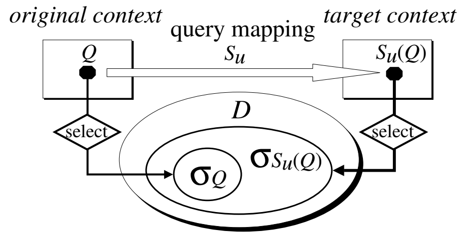

Figure 1: Conceptual illustration of query mapping.

mediator exports a faculty view fac(ln, fn, bib, dept) integrated from $aubib$ and $prof$, and a publication view pub(ti, ln, fn) from $paper$(ti, au).

Suppose that a user is looking for the papers written by some CS faculty interested in data mining. The constraint 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 *paper* and *aubib*. The join conditions $a \wedge b$ together map to x_1 : [$paper$.au = $aubib$.name]. If source T_1 does not support the proximity operator near, rather than dropping constraint c, we can relax it to (x_2:[$aubib$.bib contains data] $\wedge$ x_3:[$aubib$.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 *prof*. All the constraints except d map to *True*. Suppose that T_2 uses department code 230 for CS, thus $\mathcal{S}_2(Q)$ = [$prof$.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.

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.

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.

- 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 ∧ in Example 3). Garlic wrappers [10] support similar rewritings, but it is not clear how the mapping is done systematically.

- 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). Incidentally, while reference [12] deals with complex queries, their framework does not translate constraints.

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). (They use global relations to model data conversion; as we have discussed, constraint mapping is not symmetrical to data conversion.) In a truly heterogeneous environment, while a global vocabulary is useful for semantic integration, the sources are unlikely to agree uniformly on such a vocabulary. The source relations may have different schematic or syntactic data representations from that in the normalized world view. In addition, unless the global constraints represent the least common denominator of the source capabilities, those constraints are unlikely to be uniformly supported. Therefore, our work complements these efforts in localizing the global constraints.

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 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 [17]. Here we adapt this framework for query translation.

This section is organized as follows. To begin with, Section 4.1 presents the rule system and the associated algorithm. To focus on the essential framework, Section 4.1 considers only selection queries over a single mediator view (*e.g.*, as in Example 2). In Section 4.2 we then extend the framework slightly to handle the general case, where queries over multiple views are to be translated for multiple sources. Section 4.3 studies formally the correctness of the mapping algorithm. Finally, in Section 4.4 we discuss the complexity of the algorithm.

4.1 The Rule-Based 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 [17] 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.

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 [20] 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

[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]
$\mathcal{R}_4$)	[title contains P1] $\mapsto$ P2 = RewriteTextPat(P1); *emit*: [title-word contains P2]
$\mathcal{R}_5$)	[title = 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]
$\mathcal{R}_8$)	[keywd contains K1] $\mapsto$ K2 = RewriteTextPat(K1); *emit*: [title-word contains K2] $\vee$ [subject-word contains K2]
$\mathcal{R}_9$)	[category = C] $\mapsto$ S = MapCategoryTerms(C); *emit*: [subject = S]

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

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 *indecomposable*, *i.e.*, a rule should handle only those truly-dependent constraints. (We will formally define the notion of decomposability in Definition 2.) In other words, the mapping rules effectively encode constraint dependencies. For instance, for $\mathcal{R}_6$ the matching $\{f_y,\ f_m\}$ is indeed indecomposable. 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.*, $\mathcal{S}(f_m) = True$. Thus, $\mathcal{S}(f_y) \wedge \mathcal{S}(f_m) = \mathcal{S}(f_y) \wedge True =$ [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 indecomposable 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

Algorithm *SCM*: Simple-Conjunction Mapping
Input: • $\hat{Q}$: a simple-conjunction query in the original context.
• K: the constraint mapping specification *w.r.t.* a target system T.
Output: $\mathcal{S}(\hat{Q})$, the minimal subsuming mapping of $\hat{Q}$ *w.r.t.* T.
Procedure:

```
(1) find M(Q̂, K), all the matchings of any rule in K:
    • A ← M(Q̂, K) ≡ ∪[M(Q̂, R)], for all R ∈ K
(2) remove any matching that is a subset of other matchings:
    // i.e., submatching suppression.
    • for all m_i ∈ A:
        – for all m_j ∈ A (j ≠ i):
            – if m_j ⊆ m_i: remove m_j from A
(3) output S(Q̂) as the conjunction of all the emissions
    for the remaining matchings in A, i.e.,
    • S(Q̂) = ∏_{m∈A} S(∧(m))
```

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

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.

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.*, $\mathcal{S}(\hat{Q}_1) = a_a \wedge a_{t1} \wedge a_d \wedge (a_{t2} \vee a_{s1}) = S_1$. ■

4.2 Queries over Multiple Views across Multiple Sources

To focus on the essential framework, in the preceding discussion we assumed the simplified case of mapping selection queries over a mediator view to a source relation. As Section 2 discussed, a general integration system may export multiple views integrated from multiple sources. Unlike in the simplified case, for queries over multiple views, the general system typically supports both selection and join constraints. For the completeness of our study, this section considers the general case. As we will see, the essential framework remains effective, with only the rule system being slightly more complex.

To illustrate the issues, we will base our discussion on the integration system given in Example 3. Recall that in the example the mediator exports two views *fac* and *pub*, which are integrated from relations *paper* and *aubib* of source T_1, and relation *prof* of source T_2. In such a system, a query Q may involve multiple view instances constrained with join as well as selection conditions. Meanwhile, since the views may involve different sources, we must translate Q for each source, *i.e.*, to find the mappings $\mathcal{S}_1(Q)$ and $\mathcal{S}_2(Q)$ (as Section 2 discussed).

$K_1 = \{\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3, \mathcal{R}_4, \mathcal{R}_5\}$, for source T_1.

$\mathcal{R}_1$) [*fac*.bib contains P1] $\mapsto$ P2 = RewriteTextPat(P1); *emit*: [*fac.aubib*.bib contains P2]
$\mathcal{R}_2$) [*pub*.ti = T] $\mapsto$ *emit*: [*pub.paper*.ti = T]
$\mathcal{R}_3$) [A1 = N]; LnOrFn(A1); Value(N) $\mapsto$
A2 = AttrNameMapping(A1); *emit*: [A2 contains N]
$\mathcal{R}_4$) [AL = L]; [AF = F]; LnFnAttrs(AL, AF); Value(L); Value(F) $\mapsto$
A = AttrNameMapping(AL, AF); N = LnFnToName(L, F); *emit*: [A = N]
$\mathcal{R}_5$) [V1.ln = V2.ln]; [V1.fn = V2.fn] $\mapsto$
A1 = AttrNameMapping(V1.ln, V1.fn); A2 = AttrNameMapping(V2.ln, V2.fn)
emit: [A1 = A2]

$K_2 = \{\mathcal{R}_6, \mathcal{R}_7, \mathcal{R}_8\}$, for source T_2.

$\mathcal{R}_6$) [*fac*.A1 = N]; LnOrFn(A1); Value(N) $\mapsto$
A2 = AttrNameMapping(*fac*.A1); *emit*: [A2 = N]
$\mathcal{R}_7$) [*fac*.dept = D] $\mapsto$ C = DeptCode(D); *emit*: [*fac.prof*.dept = C]
$\mathcal{R}_8$) [*fac*[*i*].A = *fac*[*j*].A]; LnOrFn(A) $\mapsto$ *emit*: [*fac*[*i*].*prof*.A = *fac*[*j*].*prof*.A]

Figure 5: Mapping rules K_1 and K_2 respectively for source T_1 and T_2.

To perform translation, the framework requires the mapping rules for each source. To illustrate, Figure 5 gives the mapping specifications for our example. In particular, the specification K_1 consists of the rules $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$, $\mathcal{R}_4$, and $\mathcal{R}_5$ to define the mappings for source T_1. Similarly K_2 consists of the rules $\mathcal{R}_6$, $\mathcal{R}_7$, and $\mathcal{R}_8$ specifically for source T_2. Note that the specifications K_1 and K_2, as they are for different sources, are independent of each other. In other words, for a simple-conjunction query $\hat{Q}$, we will translate it separately for each source with respect to the particular mapping specification. Therefore, we will evaluate $\mathcal{S}_1(\hat{Q}) = \mathit{SCM}(\hat{Q}, K_1)$ for source T_1 and $\mathcal{S}_2(\hat{Q}) = \mathit{SCM}(\hat{Q}, K_2)$ for source T_2.

The mapping process (for each source) is essentially the same as discussed in Section 4.1. For instance, let us consider mapping the simple conjunction $\hat{Q} = a \wedge b \wedge c \wedge d$ (as given in Example 3) for source T_1. We intuitively showed the mapping in Example 3; now we more formally evaluate $\mathcal{S}_1(Q) = \mathit{SCM}(\hat{Q}, K_1)$. Matching the constraints to the rules in K_1 results in the matchings $\{c\}$ for $\mathcal{R}_1$ and $\{a, b\}$ for $\mathcal{R}_5$. Consequently, rule $\mathcal{R}_1$ will fire and yield the mapping (for source T_1) $\mathcal{S}_1(c)$ = [*fac.aubib*.bib contains data($\wedge$)mining]. Note that in the target constraint we write the source relation as *fac.aubib* to stress that it is the *aubib* expanded from the view *fac*. In general, different views may be defined with the different instances of the same source relation. Therefore, to distinguish these different relation instances we generally qualify the relation names with the containing views. For simplicity we assume here that a view does not contain multiple instances of a relation, but we note that it is straightforward to handle the situation otherwise.

In addition to the selection constraints, we must also handle the mapping of join constraints. In our example, the join constraints $\{a, b\}$ over the views *fac* and *pub* match rule $\mathcal{R}_5$ (by binding V1 to *fac* and V2 to *pub*). Referring to rule $\mathcal{R}_5$ (in Figure 5), since the attributes *fac*.ln and *fac*.fn together map to A1 = *fac.aubib*.name and in addition *pub*.ln and *pub*.fn map to A2 = *pub.paper*.au, the rule will then emit the mapping $\mathcal{S}_1(a \wedge b)$ = [*fac.aubib*.name = *pub.paper*.au]. According to Algorithm *SCM* the overall mapping is then $\mathcal{S}_1(\hat{Q}) = \mathcal{S}_1(c) \wedge \mathcal{S}_1(a \wedge b)$. Note that $\mathcal{R}_5$ is designed to handle the join constraints between any pair of the views that V1 and V2 can bind to, *i.e.*, a pair of *fac*(ln, fn, bib, dept) and *pub*(ti, ln, fn), a pair of *fac*'s, or a pair of *pub*'s.

With the essential mapping process illustrated, we next discuss the specific complications of dealing with queries over multiple views. First, in a rule it may be necessary to distinguish between selection and join constraints when a pattern can match either. For instance, rule $\mathcal{R}_3$ is designed to handle the selection constraints like [fac.ln = "Ullman"] or [pub.fn = "Hector"], since LnOrFn(A1) restricts A1 to matching only ln or fn attributes. However, note that these attributes can also participate in join constraints such as [fac.ln = pub.ln]. Therefore, without precaution the pattern [A1 = N] can unintentionally match such join constraints (*e.g.*, by binding N to pub.ln). Consequently, when it is necessary to focus on only selection constraints, we can use the condition Value(N) to restrict N to matching only values but not attributes. Similarly, we can use the condition Attr(N) to restrict the other way and thus focus on join constraints.

Second, to simplify pattern matching, we will *normalize* a join constraint if it can be written in one way or the other. In many cases, a constraint can be written as either [attr1 op1 attr2] or [attr2 op2 attr1], where op2 is the "inverse" of op1 (*e.g.*, [income > expense] ≡ [expense < income]). To avoid enumerating equivalent patterns for such constraints, we will assume one of the alternatives as the normalized representation. This normalization is simply rewriting the constraints to adopt certain operators such as ">" instead of "<" whenever possible. Otherwise, for symmetrical operators (*e.g.*, [income = expense] ≡ [expense = income]), the rewriting can assume some particular attribute ordering in the normalized representation. Note that, with this normalization, the mapping rules need only focus on the normalized representations (rather than enumerating all the alternative patterns).

Finally, we need a way to distinguish the different instances of a view in a query, if any. For instance, a query that looks for professors with the same last name may have the constraint [fac[1].ln = fac[2].ln]. Since the query involves two fac instances, we distinguish them with integer indexes. Note that this technique is similar to using tuple variables in SQL (*e.g.*, **select** ... **from** fac $F1$, fac $F2$ **where** $F1$.ln = $F2$.ln). Consequently, in a mapping rule we also use the technique to distinguish different view instances. For example, rule $\mathcal{R}_8$ in K_2 (Figure 5) designates the attribute variables as $fac[i]$.A and $fac[j]$.A. By binding $i = 1$, $j = 2$, and A = ln, the pattern can match [fac[1].ln = fac[2].ln]. Obviously we can omit the indexes (in both queries and rule patterns) when they are not necessary. For instance, the attribute fac.bib in rule $\mathcal{R}_1$ can be viewed as the abbreviation for $fac[i]$.bib and thus match the pattern for any i.

4.3 Correctness of Algorithm *SCM*

To discuss the correctness of Algorithm *SCM*, we more formally summarize our requirements for the rules individually and collectively. To begin with, we first define the notion of conjunct decomposability (and the related notion of separability) in Definition 2. Based on this notion, Definition 3 then specifies the soundness of the individual rules, as we have informally discussed.

Definition 2 (Conjunct Decomposability and Separability): A conjunction $\hat{Q} = C_1 \cdots C_n$ $(n > 1)$, where C_i's are arbitrary queries, is *decomposable* if there exist some *proper* subsets $B_1, \ldots, B_m$ of $\{C_1, \ldots, C_n\}$ (*i.e.*, $B_j \subset \{C_1, \ldots, C_n\}$), such that $\mathcal{S}(\hat{Q}) = \mathcal{S}(\wedge(B_1)) \cdots \mathcal{S}(\wedge(B_m))$. Otherwise, $\hat{Q}$ is *indecomposable*.

Furthermore, if $\hat{Q}$ can be decomposed into $\{C_1\}, \ldots, \{C_n\}$, *i.e.*, $\mathcal{S}(\hat{Q}) = \mathcal{S}(C_1) \cdots \mathcal{S}(C_n)$, then $\hat{Q}$ is also *separable*. Otherwise, $\hat{Q}$ is *inseparable* ■

In other words, when $\hat{Q}$ is decomposable, its mapping can be "synthesized" form the mappings of its proper subqueries. In addition, separability is simply a special case of decomposability, in which case the conjuncts can be individually separated. For instance, referring to Figure 2, since constraints on ti and pyear are independent, $\mathcal{S}(f_{t1} \wedge f_y) = \mathcal{S}(f_{t1}) \wedge \mathcal{S}(f_y)$, and thus $(f_{t1} \wedge f_y)$ is decomposable (in this case it is also separable). On the other hand, we cannot decompose $(f_y \wedge f_m)$ since $\mathcal{S}(f_y \wedge f_m) \neq \mathcal{S}(f_y) \wedge \mathcal{S}(f_m)$. In addition, note that the query $f_{t1} \wedge f_y \wedge f_m$ is decomposable; $\mathcal{S}(f_{t1} \wedge f_y \wedge f_m) = \mathcal{S}(f_{t1}) \wedge \mathcal{S}(f_y \wedge f_m)$. In other words, we can partition $\{f_{t1},\, f_y,\, f_m\}$ to the proper subsets $\{f_{t1}\}$ and $\{f_y, f_m\}$. (However, the query is not separable, since the set $\{f_y, f_m\}$ cannot be further separated.)

Note that whether some constraints are decomposable or not is domain specific and must be judged by human experts. Our framework captures this knowledge by requiring that each rule only deal with indecomposable constraints. (As Section 5 will discuss, disjunctions can always be separated.) Note that, with this requirement, we can tell if some constraints are interdependent by checking if they together match a rule. In other words, the mapping rules effectively encode constraint dependencies. On the other hand, we do not need rules for decomposable constraints– their mapping can be synthesized from their components (as Definition 2 requires). Thus each rule gives the minimal subsuming mapping for an indecomposable (or "atomic") set of constraints. We now summarize the soundness requirements for the mapping rules.

Definition 3 (Mapping Rule: Soundness): A mapping rule $\mathcal{R}$ is *sound* if the rule satisfies the following conditions for any matching $m = \{c_1, \ldots, c_n\}$ of $\mathcal{R}$:

- The constraint conjunction $\wedge(m) = c_1 \cdots c_n$ is indecomposable.
- The emission that $\mathcal{R}$ generates for m is the minimal subsuming mapping for the constraint conjunction. That is, $\mathcal{S}(\wedge(m)) = \mathcal{E}(\mathcal{R},\, m)$, where $\mathcal{E}(\mathcal{R},\, m)$ denotes the emission of $\mathcal{R}$ for m. ■

Furthermore, the mapping specification K (*i.e.*, the set of mapping rules collectively) must also satisfy some requirements. To begin with, we want every individual rule in K to be sound, as just defined. However, this requirement alone is not sufficient for correct mappings. In particular, an empty set of rules is trivially sound but is obviously useless. Therefore, we further specify when a rule must be present for query mapping. That is, we require that the specification be complete, *i.e.*, it has all the necessary rules. Intuitively, we need a rule to handle every indecomposable combination of constraints. For instance, for mapping to target *Amazon*, we must give the rules for the combination of constraints that are not decomposable such as ln and fn or pyear and pmonth. The specification K_{Amazon} provides $\mathcal{R}_2$ and $\mathcal{R}_6$ respectively for either case. Note that this completeness requirement implies that when giving rules we only need to focus on those "essential" combination of dependent constraints. For instance, for *Amazon* we can ignore the combinations {ln, pyear}, {ln, pmonth}, *etc.* Definition 4 formally states the soundness and completeness requirements for mapping specifications.

Definition 4 (Mapping Specification: Soundness and Completeness): A mapping specification K is *sound* if every rule in K is sound.

The specification is *complete* if for any set $\{c_1, \ldots, c_n\}$, where c_i's are arbitrary constraints that the original context supports, K has a rule matching $\{c_1, \ldots, c_n\}$ if

- the mapping for the constraint conjunction is non-trivial, *i.e.*, $\mathcal{S}(c_1 \cdots c_n) \neq \mathit{True}$, and
- the constraint conjunction $c_1 \cdots c_n$ is indecomposable. ■

To show the correctness of Algorithm *SCM*, as a basis we first present Lemma 1. The lemma states that a subconjunction (*i.e.*, a subquery X in a conjunction XY) has a "broader" mapping than that of the whole conjunction (*i.e.*, $\mathcal{S}(XY) \subseteq \mathcal{S}(X)$). This result is quite intuitive since a subconjunction is less restrictive than the conjunction (*i.e.*, $XY \subseteq X$).

Lemma 1: If $\hat{Q} = XY$, where X and Y are arbitrary queries, then $\mathcal{S}(\hat{Q}) \subseteq \mathcal{S}(X)$ (and similarly $\mathcal{S}(\hat{Q}) \subseteq \mathcal{S}(Y)$). ■

Proof: Assume that $\mathcal{S}(\hat{Q}) \not\subseteq \mathcal{S}(X)$. We will derive a contradiction that $\mathcal{S}(\hat{Q})$ cannot be the minimal subsuming mapping of $\hat{Q}$ according to Definition 1.

Let $Q' = \mathcal{S}(\hat{Q})\mathcal{S}(X)$. We now show that Q' is a better mapping for $\hat{Q}$ than $\mathcal{S}(\hat{Q})$. First, Q' is expressible in the target context, since both $\mathcal{S}(\hat{Q})$ and $\mathcal{S}(X)$ are. Second, Q' subsumes $\hat{Q}$, since both $\mathcal{S}(\hat{Q})$ and $\mathcal{S}(X)$ do: Because $X \supseteq XY$ or $X \supseteq \hat{Q}$ and (by Definition 1) $\mathcal{S}(X) \supseteq X$, it follows that $\mathcal{S}(X) \supseteq \hat{Q}$. In addition, $\mathcal{S}(\hat{Q}) \supseteq \hat{Q}$ also by Definition 1. Finally, $\mathcal{S}(\hat{Q})$ properly subsumes Q': Since $\mathcal{S}(\hat{Q}) \not\subseteq \mathcal{S}(X)$ (by assumption), it follows that $\mathcal{S}(\hat{Q}) \neq \mathcal{S}(\hat{Q})\mathcal{S}(X)$, *i.e.*, $\mathcal{S}(\hat{Q}) \supset \mathcal{S}(\hat{Q})\mathcal{S}(X)$ or $\mathcal{S}(\hat{Q}) \supset Q'$. Therefore, according to Definition 1, $\mathcal{S}(\hat{Q})$ cannot be the minimal subsuming mapping because Q' can do better, a contradiction. ∎

Finally, Theorem 1 gives the correctness of our algorithm. In other words, when the mapping specification is both sound and complete, Algorithm *SCM* will yield the minimal subsuming mapping for a simple conjunction. Note that the soundness and completeness are therefore indeed the requirements for the mapping rules that guarantee the mapping optimality.

Theorem 1 (Correctness of Algorithm *SCM*): Given a simple conjunction $\hat{Q}$ and the mapping specification K that is sound and complete for some target T, Algorithm *SCM* outputs the minimal subsuming mapping of $\hat{Q}$ with respect to T, *i.e.*, $\mathit{SCM}(\hat{Q}, K) = \mathcal{S}(\hat{Q})$. ∎

Proof: To begin with, we consider the output of the algorithm. Referring to Figure 4 but first ignoring the removal of the submatchings (*i.e.*, step 2), the output is simply the conjunction of all the emissions. Let $\hat{m}$ denote a matching as a simple conjunction, *i.e.*, $\hat{m} = \wedge(m)$. As the rules are sound, the emission for m *defines* the mapping $\mathcal{S}(\hat{m})$. The output is therefore

$$\mathit{SCM}(\hat{Q}, K) = \prod_{m \in \mathcal{M}(\hat{Q}, K)} \mathcal{S}(\hat{m}) \tag{4}$$

Furthermore, Eq. 4 will still hold with the removing of submatchings (*i.e.*, step 2 of Algorithm *SCM*). To see why, suppose that a matching m_j is removed as it is a subset of another matching m_i. In other words, $\hat{m}_j$ is a subconjunction of $\hat{m}_i$ since $m_j \subseteq m_i$. Such a submatching m_j is indeed redundant for the mapping, because by Lemma 1 $\mathcal{S}(\hat{m}_j) \supseteq \mathcal{S}(\hat{m}_i)$. Thus Eq. 4 holds through the removing of every submatching.

Based on Eq. 4, we first show that $\mathit{SCM}(\hat{Q}, K) \supseteq \mathcal{S}(\hat{Q})$ and then $\mathcal{S}(\hat{Q}) \supseteq \mathit{SCM}(\hat{Q}, K)$, and thus it follows that $\mathit{SCM}(\hat{Q}, K) = \mathcal{S}(\hat{Q})$.

(1) $\mathit{SCM}(\hat{Q}, K) \supseteq \mathcal{S}(\hat{Q})$: Referring to $\mathit{SCM}(\hat{Q}, K)$ given in Eq. 4, since every $\hat{m}$ is a subconjunction in $\hat{Q}$ (*i.e.*, $\hat{Q} = \hat{m}Y$, for some Y), according to Lemma 1, $\mathcal{S}(\hat{m}) \supseteq \mathcal{S}(\hat{Q})$. As this subsumption holds for every m in $\mathcal{M}(\hat{Q}, K)$, it follows that $\prod_{m \in \mathcal{M}(\hat{Q},K)} \mathcal{S}(\hat{m}) \supseteq \mathcal{S}(\hat{Q})$ or $\mathit{SCM}(\hat{Q}, K) \supseteq \mathcal{S}(\hat{Q})$.

(2) $\mathcal{S}(\hat{Q}) \supseteq \mathit{SCM}(\hat{Q}, K)$: Suppose that, as a simple conjunction of constraints c_i, $\hat{Q} = c_1 \cdots c_n$. Let us denote the constraints in $\hat{Q}$ as $\mathcal{C}(\hat{Q})$, *i.e.*, $\mathcal{C}(\hat{Q}) = \{c_1, \ldots, c_n\}$.

We can write $\mathcal{S}(\hat{Q}) = \prod_{v \subseteq \mathcal{C}(\hat{Q})} \mathcal{S}(\hat{v})$, *i.e.*, the conjunction of the mappings for every subset v of the constraints. To see why, note that the right hand side is the same as $\mathcal{S}(c_1 \cdots c_n) \wedge \prod_{v \subset \mathcal{C}(\hat{Q})} \mathcal{S}(\hat{v})$. Because that $\mathcal{S}(\hat{Q}) = \mathcal{S}(c_1 \cdots c_n)$ and that $\mathcal{S}(\hat{Q}) \subseteq \mathcal{S}(\hat{v})$ for every $v \subset \mathcal{C}(\hat{Q})$ (by Lemma 1), the equation follows immediately.

In other words, together with Eq. 4, we can prove that $\mathcal{S}(\hat{Q}) \supseteq \mathit{SCM}(\hat{Q}, K)$ by showing that

$$\prod_{v \subseteq \mathcal{C}(\hat{Q})} \mathcal{S}(\hat{v}) \supseteq \prod_{m \in \mathcal{M}(\hat{Q},K)} \mathcal{S}(\hat{m}) \tag{5}$$

To do so, it suffices to show that for every v in the left side, $\mathcal{S}(\hat{v})$ subsumes the right side, *i.e.*, $\mathcal{S}(\hat{v}) \supseteq \prod_{m \in \mathcal{M}(\hat{Q},K)} \mathcal{S}(\hat{m}), \forall\ v$. The proof is by induction on $|v|$, *i.e.*, the number of constraints in v.

1. $|v| = 1$, *i.e.*, $v = \{c_i\}$. We differentiate two cases: either v is a matching in $\mathcal{M}(\hat{Q}, K)$ or not.

   - if $v \in \mathcal{M}(\hat{Q}, K)$, it follows that $\mathcal{S}(\hat{v}) \supseteq \prod_{m \in \mathcal{M}(\hat{Q},K)} \mathcal{S}(\hat{m})$, since $\mathcal{S}(\hat{v})$ is among the conjuncts in the right hand side.
   - otherwise, $v \notin \mathcal{M}(\hat{Q}, K)$, *i.e.*, v is not a matching with respeact to K. In other words, K does not provide a rule for mapping v. The completeness of K (Definition 4) thus implies that either $\mathcal{S}(\hat{v}) = \mathit{True}$ or $\hat{v}$ is decomposable. The latter cannot hold since v consists of only one constraint. It follows that $\mathcal{S}(\hat{v}) = \mathit{True}$ and obviously $\mathcal{S}(\hat{v}) \supseteq \prod_{m \in \mathcal{M}(\hat{Q},K)} \mathcal{S}(\hat{m})$.

2. $|v| = k + 1$:
   As the hypothesis, assume that $\mathcal{S}(\hat{v}) \supseteq \prod_{m \in \mathcal{M}(\hat{Q},K)} \mathcal{S}(\hat{m})$ holds for all v such that $|v| \leq k$ (among those v in the left side of Eq. 5).

   We now show that the subsumption holds when $|v| = k + 1$. Similarly to case 1 for $|v| = 1$, we differentiate whether v is a matching. If $v \in \mathcal{M}(\hat{Q}, K)$, the proof is exactly the same as the preceding case. Otherwise, if v is not a matching, the completeness of the rules K implies either of the followings:

   - the mapping for $\hat{v}$ is trivial. That is, $\mathcal{S}(\hat{v}) = \mathit{True}$ and thus $\mathcal{S}(\hat{v}) \supseteq \prod_{m \in \mathcal{M}(\hat{Q},K)} \mathcal{S}(\hat{m})$.
   - Otherwise, $\hat{v}$ is decomposable, *i.e.*, there exist some proper subsets $z_1, \ldots, z_g$ of v, such that $\mathcal{S}(\hat{v}) = \mathcal{S}(\hat{z}_1) \cdots \mathcal{S}(\hat{z}_g)$. Note that as z_i is a proper subset of v and $|v| = k + 1$, it follows that $|z_i| \leq k$. Based on the induction hypothesis, $\mathcal{S}(\hat{z}_i) \supseteq \prod_{m \in \mathcal{M}(\hat{Q},K)} \mathcal{S}(\hat{m})$. Since every z_i holds this subsumption, so does $\mathcal{S}(\hat{v})$.

■

4.4 Complexity

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 decomposed in the mapping. Such

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

Figure 6: Algorithm *DNF*.

"high-degree" dependencies are obviously unlikely in practice since we can expect at least some natural schematic conventions (*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 6. 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 [21]). To name some problems, first, Algorithm *DNF* requires a *blind* DNF conversion regardless of whether some conjuncts can actually be separated. 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 7 shows a book query $\hat{Q}_{book}$ that we will use as our running example. (We will explain the shaded annotations later.) 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 8).

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. In other words, we want to find a decomposition of the conjuncts if they are decomposable (see Definition 2). 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 7) for *Amazon* with the rules K_{Amazon} (Figure 3). To begin with, since $\hat{Q}_{book}$ is conjunctive, we must figure out how to partition 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 $\mathcal{S}(\wedge(B_1)) =$ *SCM*($f_l f_f$, K_{Amazon}) $\vee$ *SCM*(f_{k1}, K_{Amazon}) $\vee$ *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,

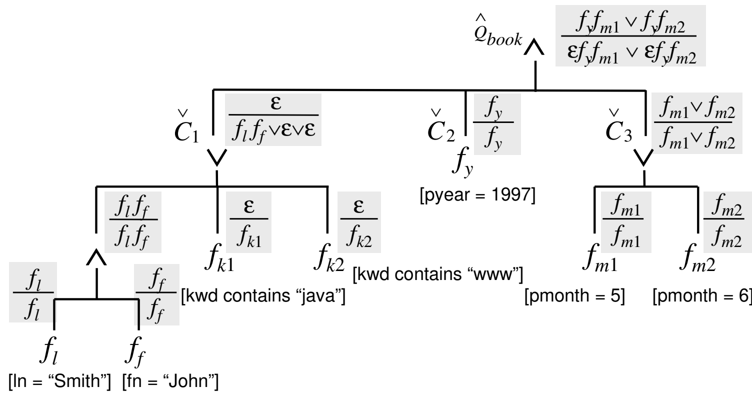

Figure 7: A query tree $\hat{Q}_{book}$. The shaded annotations illustrate the computation of EDNF.

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 8. 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. 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 8 (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 conclude Algorithm *TDQM* by giving in Theorem 2 its correctness for mapping arbitrary queries. Note that here we assume the correctness of Algorithm *PSafe*, which we will present in Section 7.

Theorem 2 (Correctness of Algorithm *TDQM*): Given an arbitrary query Q and the mapping specification K that is sound and complete for some target T, Algorithm *TDQM* outputs the minimal subsuming mapping of Q with respect to T, *i.e.*, $\mathit{TDQM}(Q, K) = \mathcal{S}(Q)$. ∎

Algorithm *TDQM***:** Top-Down Query Mapping

Input: • Q: an arbitrary query in the original context.

• K: the constraint mapping specification *w.r.t.* a target system T.

Output: $\mathcal{S}(Q)$, the minimal subsuming mapping of Q *w.r.t.* T.

Procedure:

// **Case-1: disjunctive queries**, *i.e., at an* $\vee$*-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*$(\hat{D}_i, K)$ // *recursively call TDQM for each disjunct.*

– return $\mathcal{S}(Q) \leftarrow \vee(\{\mathcal{S}(\hat{D}_1), \mathcal{S}(\hat{D}_2), \ldots, \mathcal{S}(\hat{D}_n)\})$ // *disjuncts are separable.*

// **Case-2: conjunctive queries**, *i.e., at an* $\wedge$*-node with some non-leaf children.*

• else if $Q = \wedge(\{\check{C}_1, \check{C}_2, \ldots, \check{C}_n\})$, where at least one $\check{C}_i$ is non-leaf:

– $P \leftarrow$ *PSafe*(Q, K) // *Algorithm PSafe partitions* Q *into a set of separable blocks.*

– for each $B \in P$:

– $\check{B} \leftarrow$ *Disjunctivize*$(\wedge(B))$ // *rewrite a conjunctive block into a disjunctive query.*

– $\mathcal{S}(\wedge(B)) \leftarrow$ *TDQM*$(\check{B}, K)$ // *recursively call TDQM for disjunctive query* $\check{B}$.

– return $\mathcal{S}(Q) \leftarrow \wedge_{B \in P} \mathcal{S}(\wedge(B))$

// **Case-3: simple conjunctions**, *i.e., at a leaf or an* $\wedge$*-node with only leaf children.*

• else if Q is a simple conjunction with one or more constraints:

– return $\mathcal{S}(Q) \leftarrow$ *SCM*(Q, K) // *call Algorithm SCM with inputs* Q *and* K.

Function *Disjunctivize*$(\wedge(B))$: // *rewrite a conjunctive query* $\wedge(B)$ *into a disjunctive form* $\check{B}$.

// B *is a set of* k *conjuncts, i.e.,* $B = \{\check{C}_1, \check{C}_2, \ldots, \check{C}_k\}$, *and* $\check{C}_i = \vee(\{\hat{D}_{i1}, \hat{D}_{i2}, \ldots, \hat{D}_{im_i}\})$.

• if $k = 1$: return $\check{B} \leftarrow \check{C}_1$ // *no need to rewrite single-conjunct block.*

• else: // *if more than one conjunct, rewrite* $\wedge(B)$ *by distributing* $\wedge$ *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 $\check{B} \leftarrow \vee(\{\hat{D}_{1j_1}\hat{D}_{2j_2} \cdots \hat{D}_{kj_k} \mid j_i \in [1 : m_i]\})$

Figure 8: Algorithm *TDQM* for mapping arbitrary queries.

Proof: We differentiate between the case when Q is a simple conjunction and otherwise. First, Algorithm *TDQM* does handle simple conjunction correctly: Referring to Figure 8 (Case-3), when Q is a simple conjunction, the algorithm simply processes Q using Algorithm *SCM*, *i.e.*, *TDQM*$(Q, K) =$ *SCM*(Q, K). As K is sound and complete, it follows from Theorem 1 that *TDQM*$(Q, K) = \mathcal{S}(Q)$.

We next consider when Q is an arbitrary query other than a simple conjunction. Assuming that Algorithm *TDQM* does not handle Q correctly, we will show that the algorithm must also fail with some simple conjunction, thus a contradiction to the preceding case.

To begin with, for any Q such that *TDQM*$(Q, K) \neq \mathcal{S}(Q)$, we can derive from Q a *subquery* Q' that Algorithm *TDQM* cannot map correctly either. Compared to Q, such a subquery Q' is *strictly simpler* in the sense that its query tree is a *proper subgraph* of that of Q. In other words, Q' consists of only some (but not all) of the subtrees in Q. Since Q is either conjunctive or disjunctive, we discuss how to find such a subquery Q' for either case:

1. If Q is disjunctive, *i.e.*, $Q = \vee(\{\hat{D}_1, \ldots, \hat{D}_n\})$ and $n > 1$. As discussed in Section 5, disjuncts are always separable, *i.e.*, $\mathcal{S}(Q) = \mathcal{S}(\hat{D}_1) \vee \cdots \vee \mathcal{S}(\hat{D}_n)$ (see reference [15, 16]) . For such queries, referring to Figure 8 (Case-1), our algorithm outputs *TDQM*$(Q, K) =$ *TDQM*$(\hat{D}_1, K) \vee \cdots \vee$ *TDQM*$(\hat{D}_n, K)$. If *TDQM*$(Q, K) \neq \mathcal{S}(Q)$, there must exist some disjunct $\hat{D}_i$ such that *TDQM*$(\hat{D}_i, K) \neq \mathcal{S}(\hat{D}_i)$. Let $Q' = \hat{D}_i$. Note that Q' is strictly simpler than Q: In terms of the query trees, Q' is a subgraph of Q, since Q' consists of only one of the n ($n > 1$) disjunct subtrees in Q.

2. Otherwise, if Q is conjunctive (but not a simple conjunction), *i.e.*, $Q = \wedge(\{\check{C}_1, \ldots, \check{C}_n\})$ and $n > 1$.

Let $\{B_1, \ldots, B_m\}$ be the partition returned by Algorithm *PSafe*. As we will see in Theorem 6, the partition guarantees that $\mathcal{S}(Q) = \mathcal{S}(\wedge(B_1)) \cdots \mathcal{S}(\wedge(B_m))$. Let $\breve{B}_j$ = *Disjunctivize*$(\wedge(B_j))$. Referring to Figure 8, it is obvious that $\breve{B}_j \equiv \wedge(B_j)$ and thus $\mathcal{S}(Q) = \mathcal{S}(\breve{B}_1) \cdots \mathcal{S}(\breve{B}_m)$. In comparison, our algorithm outputs (via Case-2) *TDQM*(Q, K) = *TDQM*$(\breve{B}_1, K) \vee \cdots \vee$ *TDQM*$(\breve{B}_m, K)$. For $\mathcal{S}(Q)$ and *TDQM*(Q, K) to be different, it is required that *TDQM*$(\breve{B}_j, K) \neq \mathcal{S}(\breve{B}_j)$, for some B_j.

Let's now focus on such $\breve{B}_j$. Note that $\breve{B}_j \equiv \wedge(B_j)$. Since $\wedge(B_j)$ covers some or all of the conjuncts in Q, $\wedge(B_j)$ or $\breve{B}_j$ must be either strictly simpler than or equivalent to Q. (The latter can happen when B_j consists of all the conjuncts in Q.) Furthermore, as $\breve{B}_j$ is disjunctive, we then apply the preceding case to separate a subquery Q' that is strictly simpler than $\breve{B}_j$ and in turn also strictly simpler than Q.

Repeating this process recursively, each time we start with the subquery Q' just identified to find an even simpler subquery, and so on. Thus the sequence of queries so derived are "monotonically" strictly-simpler. Given a query Q with a finite parse tree, eventually we will end up with a simple conjunction Q' (where neither Case 1 nor Case 2 can apply) such that *TDQM*$(Q', K) \neq \mathcal{S}(Q')$. Thus we reach a contradiction since we have shown that Algorithm *TDQM* processes simple conjunctions correctly. ∎

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 Partitioning

This section discusses how we can partition the conjuncts in 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). In Section 7.2 we then present 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). (See Definition 2 for the formal definition of separability.) 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 (which assume alternating $\wedge$ and $\vee$), it is the basis for the general case.

7.1.1 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 *SCM*$(\hat{Q}, K)$ = *SCM*$(\hat{C}_1, K) \cdots$ *SCM*$(\hat{C}_n, K)$. As Algorithm *SCM* is essentially a rule matching process, if all the matchings in *SCM*$(\hat{Q}, K)$ can also be found in some *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 5 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 7). 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 5 (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 this 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, for the completeness of our study, in what follows we present the precise (*i.e.*, sufficient and necessary) separability condition for the base-case conjunctions (Theorem 3). It follows immediately from the precise condition that the safety in Definition 5 does imply separability (Corollary 1). Moreover, we note 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 5 will be adequate. If we use Definition 5 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.

Precise Separability Condition

We just observed in Example 7 that a cross-matching m spanning across some $\hat{C}_i$'s can make $\hat{Q}$ inseparable. However, $\hat{Q}$ is truly inseparable only if the cross-matching is *essential* such that omitting it would result in a non-minimal mapping.

To illustrate, consider Example 7 again. The mapping $\mathcal{S}(\hat{C}_1)\mathcal{S}(\hat{C}_2)\mathcal{S}(\hat{C}_3)$ resulted from conjunct separation is not minimal for $\hat{Q}$: The cross-matching $m = \{f_y, f_{m1}\}$ can further contribute to the mapping. In other words, $\mathcal{S}(\hat{C}_1)\mathcal{S}(\hat{C}_2)\mathcal{S}(\hat{C}_3)\mathcal{S}(\wedge(m))$ is more selective because $\mathcal{S}(\hat{C}_1)\mathcal{S}(\hat{C}_2)\mathcal{S}(\hat{C}_3) \not\subseteq \mathcal{S}(\wedge(m))$– We thus determine that m is essential for the mapping. With this notion, a conjunction is separable if there is no essential cross-matchings. We formally state this separability criterion in Theorem 3.

Theorem 3 (Separability for Base-Case Conjunctions): Let $\hat{Q} = \hat{C}_1 \cdots \hat{C}_n$, where $\hat{C}_i$'s are simple conjunctions. Let $\delta = \mathcal{M}(\hat{Q}, K) - \cup_{i=1}^{n}\mathcal{M}(\hat{C}_i, K)$ for some rules K. $\hat{Q}$ is *separable w.r.t.* K if and only if for every matching $m \in \delta$,

$$\mathcal{S}(\hat{C}_1) \cdots \mathcal{S}(\hat{C}_n) \subseteq \mathcal{S}(\wedge(m)). \quad (6)$$ ∎

Proof: (if) Since $\hat{C}_1, \ldots, \hat{C}_n$ as well as $\hat{Q}$ are all simple conjunctions, their minimal subsuming mappings can be generated with Algorithm *SCM* (Theorem 1). Thus, referring to Eq. 4 discussed in Theorem 1, the mapping of $\hat{Q}$ is $\mathcal{S}(\hat{Q}) = \mathit{SCM}(\hat{Q}, K) = \prod_{m \in \mathcal{M}(\hat{Q},K)} \mathcal{S}(\hat{m})$. Note that $\mathcal{M}(\hat{Q}, K) = \mathcal{M}(\hat{C}_1, K) \cup \cdots \cup \mathcal{M}(\hat{C}_n, K) \cup \delta$. Therefore,

$$\mathcal{S}(\hat{Q}) = \textstyle\prod_{m \in \mathcal{M}(\hat{C}_1,K)} \mathcal{S}(\hat{m}) \cdots \prod_{m \in \mathcal{M}(\hat{C}_n,K)} \mathcal{S}(\hat{m}) \prod_{m \in \delta} \mathcal{S}(\hat{m}).$$

Similarly, for each $\hat{C}_i$, the mapping is $\mathcal{S}(\hat{C}_i) = \mathit{SCM}(\hat{C}_i, K) = \prod_{m \in \mathcal{M}(\hat{C}_i,K)} \mathcal{S}(\hat{m})$. Thus the mapping of $\hat{Q}$ can be written as

$$\mathcal{S}(\hat{Q}) = \mathcal{S}(\hat{C}_1) \cdots \mathcal{S}(\hat{C}_n) \wedge \textstyle\prod_{m \in \delta} \mathcal{S}(\hat{m}) \quad (7)$$

Obviously if $\mathcal{S}(\hat{C}_1) \cdots \mathcal{S}(\hat{C}_n) \subseteq \mathcal{S}(\hat{m})$ for every cross-matching $m \in \delta$, then $\mathcal{S}(\hat{Q}) = \mathcal{S}(\hat{C}_1) \cdots \mathcal{S}(\hat{C}_n)$, *i.e.*, $\hat{Q}$ is separable.

(only if) Suppose that there exist matchings $x_1, \ldots, x_l$ in δ such that $\mathcal{S}(\hat{C}_1)\cdots\mathcal{S}(\hat{C}_n) \not\subseteq \mathcal{S}(\hat{x}_i)$. We can then write Eq. 7 as $\mathcal{S}(\hat{Q}) = \mathcal{S}(\hat{C}_1)\cdots\mathcal{S}(\hat{C}_n) \wedge [\mathcal{S}(\hat{x}_1)\cdots\mathcal{S}(\hat{x}_l)]$. Consequently $\hat{Q}$ is not separable, because $\mathcal{S}(\hat{Q}) \subset \mathcal{S}(\hat{C}_1)\cdots\mathcal{S}(\hat{C}_n)$ and thus the right side is not the minimal subsuming mapping. ■

Operationally, to test separability with Theorem 3, we first evaluate δ, all the *cross-matchings* found with $\hat{Q}$ but not with any individual $\hat{C}_i$'s. For any such matching m, we check with Eq. 6 if m can actually contribute to the mapping of $\hat{Q}$. If $\mathcal{S}(\wedge(m))$ subsumes $\mathcal{S}(\hat{C}_1)\mathcal{S}(\hat{C}_2)\cdots\mathcal{S}(\hat{C}_n)$, then omitting m has no impact on $\mathcal{S}(\hat{Q})$. We next illustrate the usage of Theorem 3 with Example 8.

Example 8: Suppose that a source G supports map queries that specify rectangle areas. Specifically, queries are in terms of constraints on attributes X_{range}, Y_{range} (respectively the range of x and y coordinate), C_{ll} and C_{ur} (respectively the lower-left and upper-right coordinate). For instance, a query selecting an area bounded by $x = 10$, $x = 30$, $y = 20$, and $y = 40$ can be written as either g_1:[X_{range} = (10:30)] $\wedge$ g_2:[Y_{range} = (20:40)] or equivalently g_3:[C_{ll} = (10,20)] $\wedge$ g_4:[C_{ur} = (30,40)]. Figure 9 illustrates this area as the dark rectangle labeled g_1g_2. Note that g_3 by itself specifies an open area with boundary $x = 10$ and $y = 20$, *i.e.*, the whole shaded area (including that of g_1g_2).

Assume a mediator F supports the same map queries but through different attributes. Specifically, F uses attributes x_{min} and x_{max} for respectively the lower and upper bound of x coordinate, and similarly y_{min} and y_{max}. Therefore, the above query can be expressed in F as f_1:[x_{min} = 10] $\wedge$ f_2:[x_{max} = 30] $\wedge$ f_3:[y_{min} = 20] $\wedge$ f_4:[y_{max} = 40]. Consequently, for translation from F to G, we would compose a specification K with rules handling the mapping from the conjunction $\mathsf{x}_{min} \wedge \mathsf{x}_{max}$ to X_{range}, $\mathsf{y}_{min} \wedge \mathsf{y}_{max}$ to Y_{range}, $\mathsf{x}_{min} \wedge \mathsf{y}_{min}$ to C_{ll}, and $\mathsf{x}_{max} \wedge \mathsf{y}_{max}$ to C_{ur}.

First, consider an F query $\hat{Q} = \hat{C}_1 \wedge \hat{C}_2 = (f_1f_2)(f_3f_4)$. We want to test if $\hat{Q}$ is separable with Theorem 3. Matching the (sub-) queries against K we obtain $\mathcal{M}(\hat{C}_1, K) = \{m_1{:}\{f_1, f_2\}\}$, $\mathcal{M}(\hat{C}_2, K) = \{m_2{:}\{f_3, f_4\}\}$, and $\mathcal{M}(\hat{Q}, K)$ includes m_1, m_2, plus $m_3 = \{f_1, f_3\}$, and $m_4 = \{f_2, f_4\}$. There are thus two cross-matchings, *i.e.*, $\delta = \mathcal{M}(\hat{Q}, K) - (\mathcal{M}(\hat{C}_1, K) \cup \mathcal{M}(\hat{C}_2, K)) = \{m_3, m_4\}$.

Although we have identified two cross-matchings, $\hat{Q}$ is in fact separable because both matchings are "redundant." We test this redundancy with with Eq. 6. In other words, we verify if $\mathcal{S}(\hat{C}_1)\mathcal{S}(\hat{C}_2) \subseteq \mathcal{S}(\wedge(m_3))$ (and similarly for m_4). This subsumption condition indeed holds: $\mathcal{S}(\hat{C}_1)\mathcal{S}(\hat{C}_2) = \mathcal{S}(f_1f_2)\mathcal{S}(f_3f_4) = g_1g_2$ (the corresponding x and y ranges), which is subsumed by $\mathcal{S}(\wedge(m_3)) = g_3$ (the lower-left coordinate). To illustrate this subsumption, Figure 9 shows that the point (50,30) is in g_3 but not g_1g_2. Similarly, Eq. 6 also holds for m_4. Since no cross-matchings are essential, $\hat{Q}$ is indeed separable.

As a second example, let's consider $\hat{Q} = \hat{C}_1 \wedge \hat{C}_2 = (f_1f_4)(f_2f_3)$. In this case $\mathcal{M}(\hat{C}_1, K) = \phi$, *i.e.*, there is no mapping for either f_1 and f_4 individually, or together. Similarly, $\mathcal{M}(\hat{C}_2, K) = \phi$. However, $\mathcal{M}(\hat{Q}, K)$ is the same as the previous example, *i.e.*, $\{m_1, m_2, m_3, m_4\}$. Thus we have four cross-matchings: $\delta = \{m_1, m_2, m_3, m_4\}$.

In this case, $\hat{Q}$ is not separable because some (in fact all) cross-matchings are essential; they fails to satisfy the condition in Theorem 3, *e.g.*, $\mathcal{S}(\hat{C}_1)\mathcal{S}(\hat{C}_2) \not\subseteq \mathcal{S}(\wedge(m_3))$. In fact, both $\mathcal{S}(\hat{C}_i)$'s are *True* because there is no matching for either $\hat{C}_1$ or $\hat{C}_2$. The left side of the condition is therefore *True*, while the right side is $\mathcal{S}(\wedge(m_3)) = g_3$. Obviously, *True* $\not\subseteq g_3$, thus the condition fails. ■

The precise conditions of Theorem 3 can be expensive to apply. Although finding cross-matchings is straightforward, testing the subsumption condition Eq. 6 for such matchings can be costly or even impossible. As Example 8 shows, we need to find $\mathcal{S}(\wedge(m))$ as well as $\mathcal{S}(\hat{C}_i)$'s simply for testing separability. Moreover, testing the subsumption condition itself is very domain-specific and is hard to automate, if possible at all.

We therefore give the notion of safety in Definition 5 as a practical measure for determining separability. It follows immediately from Theorem 3 that safety is a *sufficient* condition (in which case $\delta = \phi$) that guarantees

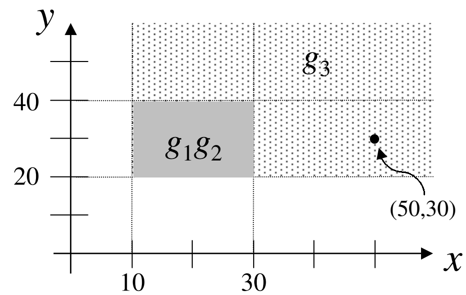

Figure 9: Query g_3 (all the shaded area) subsumes query g_1g_2 (the dark rectangle).

separability, as formally stated in Corollary 1. Intuitively, Definition 5 omits the test of essentialness and only checks the existence of cross-matchings. In most cases, we believe a redundant cross-matching is rare, and the simple test of safety will be adequate. Note that Example 8 is an exception since source G supports inter-dependent attributes (*i.e.*, a pair of X_{range} and Y_{range} is equivalent to a pair of C_{ll} and C_{ur}), which we believe to be unusual in practice.

Corollary 1: Let $\hat{Q} = \hat{C}_1 \cdots \hat{C}_n$, where $\hat{C}_i$'s are simple conjunctions. If $\hat{Q}$ is safe according to Definition 5, then $\hat{Q}$ is separable. ■

7.1.2 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 inter-dependencies 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 9.

Example 9: 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 8). 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 9 suggests the following safety condition. Note that Definition 6 defines safety recursively; as we will see, Definition 5 is the base case that grounds the recursion.

Definition 6 (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 $\mathcal{S}(x) = \mathit{True}$ 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 $\mathcal{S}(x) = \mathit{True}$. 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 $\mathcal{S}(x) \neq \mathit{True}$.

To explain the anomalies and to show the correctness of Definition 6, we next discuss the precise separability condition for the general-case conjunctions. However, such anomalies should be rare in practice. That is, we believe that we can use Definition 6, 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.

Precise Separability Condition

We just observed the anomaly that $\mathcal{S}((x \vee y)(z)) = \mathcal{S}(x \vee y)\mathcal{S}(z)$ even if $\mathcal{S}(yz) \neq \mathcal{S}(y)\mathcal{S}(z)$ (*i.e.*, $\mathcal{S}(yz) \subset \mathcal{S}(y)\mathcal{S}(z)$). Note that because $\mathcal{S}(x) = \mathit{True}$, $\mathcal{S}(xz) = \mathit{True} \wedge \mathcal{S}(z) = \mathcal{S}(z)$. Furthermore, since $\check{Q}$ has two terms xz and yz, the "difference" between $\mathcal{S}(yz)$ and $\mathcal{S}(y)\mathcal{S}(z)$, *i.e.*, $[\mathcal{S}(y)\mathcal{S}(z)]\ \neg\mathcal{S}(yz)$, is subsumed (or "absorbed") by the other term $\mathcal{S}(xz)$, *i.e.*, $[\mathcal{S}(y)\mathcal{S}(z)]\ \neg\mathcal{S}(yz) \subseteq \mathcal{S}(xz)$. The subsumption holds because the left side is a conjunction with $\mathcal{S}(z)$ and the right is simply $\mathcal{S}(z)$. Therefore, this subsumption cancels the effect of the unsafe terms yz.

The above observation suggests that $\hat{Q}$ is separable if and only if any disjunct in $\mathit{Disjunctivize}(\hat{Q})$ is either separable, or else its effect can be canceled by the mappings of the other terms. Theorem 4 formally gives this precise condition for separability.

Theorem 4 (Separability 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 separable *w.r.t.* rules K if and only if for every disjunct $\hat{D}_j = I_{1k_1} \cdots I_{nk_n}$ in $\check{Q}$,

$$[\mathcal{S}(I_{1k_1}) \cdots \mathcal{S}(I_{nk_n})]\ \neg\mathcal{S}(\hat{D}_j) \subseteq \textstyle\sum_{\hat{D}_{j'} \in \check{Q}, j' \neq j} \mathcal{S}(\hat{D}_{j'}). \tag{8}$$ ■

Proof: (if) We first show that if Eq. 8 holds for every $\hat{D}_j$, then $\hat{Q}$ is separable. To begin with, the mapping of $\hat{Q}$ can be computed with $\check{Q}$ as

$$\mathcal{S}(\hat{Q}) \equiv \mathcal{S}(\check{Q}) = \textstyle\sum_{\hat{D}_j \in \check{Q}} \mathcal{S}(\hat{D}_j). \tag{9}$$

For simplicity, for each $\hat{D}_j = I_{1k_1} \cdots I_{1k_n}$, let Z_j denote $\mathcal{S}(I_{1k_1}) \cdots \mathcal{S}(I_{nk_n})$. The left side of Eq. 8 can then be written as $Z_j \neg\mathcal{S}(\hat{D}_j)$. Note that in general by Lemma 1,

$$\mathcal{S}(\hat{D}_j) \subseteq Z_j. \tag{10}$$

If $\hat{D}_j$ is separable, then $\mathcal{S}(\hat{D}_j) = Z_j$.

Suppose that Eq. 8 holds for every $\hat{D}_j$. Since the right side of Eq. 8 is subsumed by that of Eq. 9, by transitivity it follows that $Z_j \neg\mathcal{S}(\hat{D}_j) \subseteq \mathcal{S}(\hat{Q})$. As this subsumption holds for every $\hat{D}_j$, we can write $\mathcal{S}(\hat{Q})$ as $\mathcal{S}(\hat{Q}) \vee \sum_{\hat{D}_j \in \check{Q}} Z_j \neg\mathcal{S}(\hat{D}_j)$ or (by Eq. 9) as $\sum_{\hat{D}_j \in \check{Q}} \mathcal{S}(\hat{D}_j) \vee \sum_{\hat{D}_j \in \check{Q}} Z_j \neg\mathcal{S}(\hat{D}_j)$. In addition, since $\mathcal{S}(\hat{D}_j) \subseteq Z_j$ (see Eq. 10), we can substitute $Z_j\mathcal{S}(\hat{D}_j)$ for $\mathcal{S}(\hat{D}_j)$ in the first term, *i.e.*,

$$\mathcal{S}(\hat{Q}) = \textstyle\sum_{\hat{D}_j \in \check{Q}} Z_j\mathcal{S}(\hat{D}_j) \vee \sum_{\hat{D}_j \in \check{Q}} Z_j \neg\mathcal{S}(\hat{D}_j) = \sum_{\hat{D}_j \in \check{Q}} Z_j[\mathcal{S}(\hat{D}_j) \vee \neg\mathcal{S}(\hat{D}_j)] = \sum_{\hat{D}_j \in \check{Q}} Z_j$$

Meanwhile, we can show that $\mathcal{S}(\check{C}_1)\cdots\mathcal{S}(\check{C}_n)$ also results in the last expression, and thus $\hat{Q}$ is separable:

$$\begin{aligned}\mathcal{S}(\check{C}_1)\cdots\mathcal{S}(\check{C}_n) &= \mathcal{S}(I_{11}\vee\cdots\vee I_{1m_1})\cdots\mathcal{S}(I_{n1}\vee\cdots\vee I_{nm_n})\\ &= [\mathcal{S}(I_{11})\vee\cdots\vee\mathcal{S}(I_{1m_1})]\cdots[\mathcal{S}(I_{n1})\vee\cdots\vee\mathcal{S}(I_{nm_n})]\\ &= \textstyle\sum_{I_{1k_1}\cdots I_{nk_n}\text{in }\check{Q}}\mathcal{S}(I_{1k_1})\cdots\mathcal{S}(I_{nk_n}) = \sum_{\hat{D}_j\in\check{Q}}Z_j \end{aligned} \tag{11}$$

(only if) We show that $\hat{Q}$ is not separable, if some $\hat{D}_h$ fails to satisfy Eq. 8, *i.e.*,

$$Z_h\neg\mathcal{S}(\hat{D}_h)\not\subseteq\textstyle\sum_{\hat{D}_{j'}\in\check{Q},j'\neq h}\mathcal{S}(\hat{D}_{j'}). \tag{12}$$

As an aside, it is straightforward to see that, in terms of set algebra, if $A\not\subseteq B$ and $A\cap C=\phi$, then $A\cup C\not\subseteq B\cup C$. Based on this property, since $Z_h\neg\mathcal{S}(\hat{D}_h)\cap\mathcal{S}(\hat{D}_h)=\phi$, it follows Eq. 12 that

$$\mathcal{S}(\hat{D}_h)\vee Z_h\neg\mathcal{S}(\hat{D}_h)\not\subseteq\mathcal{S}(\hat{D}_h)\vee\textstyle\sum_{\hat{D}_{j'}\in\check{Q},j'\neq h}\mathcal{S}(\hat{D}_{j'}) \tag{13}$$

Furthermore, because $\mathcal{S}(\hat{D}_h)\subseteq Z_h$ and thus $\mathcal{S}(\hat{D}_h)=Z_h\mathcal{S}(\hat{D}_h)$, the left side of Eq. 13 equals $Z_h\mathcal{S}(\hat{D}_h)\vee Z_h\neg\mathcal{S}(\hat{D}_h)=Z_h$. Note also that the right side is simply $\mathcal{S}(\hat{Q})$ as given in Eq. 9. Therefore the equation becomes $Z_h\not\subseteq\mathcal{S}(\hat{Q})$. In contrast, since Z_h is among the disjuncts in Eq. 11, it follows that $Z_h\subseteq\mathcal{S}(\check{C}_1)\cdots\mathcal{S}(\check{C}_n)$. By comparison we conclude that $\mathcal{S}(\hat{Q})\neq\mathcal{S}(\check{C}_1)\cdots\mathcal{S}(\check{C}_n)$, *i.e.*, $\hat{Q}$ is not separable. ∎

Note that, if a disjunct $\hat{D}_i$ is safe (and thus separable), we do not need to test the subsumption condition Eq. 8 at all. In this case, because $\mathcal{S}(\hat{D}_i)=\mathcal{S}(I_{1k_1})\mathcal{S}(I_{2k_2})\cdots\mathcal{S}(I_{nk_n})$, Eq. 8 holds trivially because the left side is simply ϕ. In fact, this case is exactly the safety condition presented in Definition 6. Thus the correctness of Definition 6 follows immediately.

Corollary 2: Let $\hat{Q}=\check{C}_1\cdots\check{C}_n$, where $\check{C}_i$'s are disjunctive queries. If $\hat{Q}$ is safe according to Definition 6, then $\hat{Q}$ is separable. ∎

When $\hat{D}_i$ is unsafe, Theorem 4 further verifies that such unsafe terms do have an impact on the overall mapping. In other words, if Eq. 8 holds, then we can ignore such unsafe terms. Note that this subsumption condition is hard and expensive to verify. In practice, we believe that it is adequate to use the safety condition instead. Definition 6 is much cheaper to test, and in most cases unsafe conjunctions are simply inseparable.

7.1.3 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 6. 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 5.

In fact, we can first convert $\check{C}_i$'s into DNF to avoid the recursion: Note that, in Definition 6, 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 5. We illustrate this process with Example 10.

Example 10: Consider (in Figure 7) $\hat{Q}_{book}=\check{C}_1\check{C}_2\check{C}_3=(f_l f_f\vee f_{k1}\vee f_{k2})(f_y)(f_{m1}\vee f_{m2})$. Note that $\check{C}_i$'s are already in DNF. We show that $\hat{Q}_{book}$ is unsafe. According to Definition 6, we need the disjunctive form of $\hat{Q}_{book}$:

$$\begin{aligned}\check{Q}_{book}=\textit{Disjunctivize}(\hat{Q}_{book})=\vee(\{\ &\hat{D}_1{:}(f_l f_f)(f_y)(f_{m1}),\hat{D}_2{:}(f_l f_f)(f_y)(f_{m2}),\hat{D}_3{:}(f_{k1})(f_y)(f_{m1}),\\ &\hat{D}_4{:}(f_{k1})(f_y)(f_{m2}),\hat{D}_5{:}(f_{k2})(f_y)(f_{m1}),\hat{D}_6{:}(f_{k2})(f_y)(f_{m2})\})\end{aligned} \tag{14}$$

We then need to check if $\hat{D}_i$'s are safe. For this check we can apply Definition 5 since $\hat{D}_i$'s are simple-conjunction conjunctions. (If $\check{C}_i$'s were not in DNF, we would have to use Definition 6 recursively.) In this case all $\hat{D}_i$'s are unsafe: There is a cross-matching (for rules K_{Amazon}) $\{f_y, f_{m1}\}$ in $\hat{D}_1$, $\hat{D}_3$, and $\hat{D}_5$, and $\{f_y, f_{m2}\}$ in the others. Therefore, $\hat{Q}_{book}$ is unsafe. ■

However, this "brute-force" approach is not as efficient as possible; it unnecessarily relies on $\hat{C}_i$'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 EDNF), 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 preceding example.

Example 11 (Essential DNF): Consider again (in Figure 7) $\hat{Q}_{book} = \check{C}_1\check{C}_2\check{C}_3 = (f_l f_f \vee f_{k1} \vee f_{k2})(f_y)(f_{m1} \vee f_{m2})$. As we will see, the EDNF'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 EDNF. (We will later explain how to do "prematching" to find the potential matchings.) 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 in Example 10, because using $\check{C}_i$'s EDNF 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. ■

The major challenge remaining is therefore how to find the EDNF expression. We illustrate this with query $\hat{Q}_{book}$. Essentially, we want to remove those useless constraints that will never participate in a cross-matching. Therefore, the first step is to find all the matchings that *may* be involved in mapping $\hat{Q}_{book}$. (We cannot know the exact matchings until we actually process the whole query.) In particular, we simply match the rules with all the constraints, regardless of how they actually appear in $\hat{Q}_{book}$. In other words, let $\mathcal{C}(\hat{Q}_{book})$ denote the constraints in $\hat{Q}_{book}$, *i.e.*, $\mathcal{C}(\hat{Q}_{book}) = \{f_y, f_{m1}, f_{m2}, f_l, f_f, f_{k1}, f_{k2}\}$. We first find the *potential matchings*: $M_p = \mathcal{M}(\mathcal{C}(\hat{Q}), K_{Amazon}) = \{\{f_y, f_{m1}\}, \{f_y, f_{m2}\}, \{f_y\}, \{f_l\}, \{f_l, f_f\}, \{f_{k1}\}, \{f_{k2}\}\}$.

We next discuss how we actually compute the EDNF expression of a query. Note that, since a query tree may have several $\wedge$-nodes that require safety checks, we need EDNF for all of them. Therefore, our process will compute the EDNF expressions for all nodes (representing subqueries) in the given tree "at once." Intuitively, note that we can compute the (ordinary) DNF of a query tree in a bottom-up process: Starting from the leaves, at each node we obtain the DNF (for the subquery rooted there) by merging the children's DNF in a disjunctive form. For instance, in Example 10, $\check{Q}_{book}$ is actually the DNF of $\hat{Q}_{book}$ obtained this way from $\check{C}_i$'s DNF.

Similarly, we can also compute EDNF in such a bottom-up process. In addition, as will be clear, during the process we remove terms as soon as they become useless. To illustrate, in Figure 7 we annotate each node with $\mathcal{D}_e(R)$ and $\mathcal{D}(R)$ for the subquery R rooted there. The annotations are written as $\mathcal{D}_e(R)/\mathcal{D}(R)$ in the shaded boxes. Note that, as in Example 11, $\mathcal{D}(Q)$ denotes the DNF of Q based on the EDNF of its (immediate) subqueries. Starting from the leaves, at each node (for subquery R), we first compute $\mathcal{D}(R)$ from the children's EDNF, and then remove useless terms to obtain its own EDNF. This recursive process effectively streamlines the computation of $\mathcal{D}(\cdot)$ and $\mathcal{D}_e(\cdot)$.

To illustrate, consider the subtree $\check{C}_1$ in Figure 7; we show that $\mathcal{D}_e(\check{C}_1) = \epsilon$. Initially, $\mathcal{D}(\cdot)$ for a leaf node is simply the constraint itself, *i.e.*, $\mathcal{D}(f_l) = f_l$, $\mathcal{D}(f_{k1}) = f_{k1}$, *etc.* Note that f_{k1} is in fact useless since it can only contribute to the matching $\{f_{k1}\}$ by itself, as indicated by M_p. Thus we can delete f_{k1} (and similarly for f_{k2}) from $\mathcal{D}(f_{k1})$ and obtain $\mathcal{D}_e(f_{k1}) = \epsilon$. (The ϵ symbol serves as a place holder simply for the ease of computation.) In contrast, we cannot delete f_l (and similarly for f_f), because M_p contains a (potential) cross-matching $\{f_l, f_f\}$ that f_l can contribute to (with other terms outside the current subtree).

For subquery $f_l \wedge f_f$, based on the children's EDNF, we compute $\mathcal{D}(f_l f_f) = \mathcal{D}_e(f_l)\mathcal{D}_e(f_f) = f_l f_f$. It appears that we can delete the whole term $f_l f_f$, because it *fully contains* the only relevant matchings $\{f_l\}$ and $\{f_l, f_f\}$ in M_p, and thus cannot further combine with other terms to form a cross-matching. However, deleting $f_l f_f$ at this point can result in a *false-positive* cross-matchings: To see this, consider the conjunction $\hat{Q} = (f_l f_f)(f_l)(f_f)$. Since $\{f_l, f_f\}$ is fully contained in the first conjunct, it is not a cross-matching. However, if we delete $f_l f_f$ and $\hat{Q}$ becomes $(\epsilon)(f_l)(f_f)$, then we would adversely identify $\{f_l, f_f\}$ as a matching across conjuncts (f_l) and (f_f).

Moving up one level, we reach the node $\check{C}_1$ for subquery $(f_l f_f \vee f_{k1} \vee f_{k2})$. We first compute $\mathcal{D}(\check{C}_1)$ as the disjunction of its children's EDNF: $\mathcal{D}(\check{C}_1) = \mathcal{D}_e(f_l f_f) \vee \mathcal{D}_e(f_{k1}) \vee \mathcal{D}_e(f_{k2}) = f_l f_f \vee \epsilon \vee \epsilon$. At this point we can actually delete $f_l f_f$, the false positive just illustrated will not occur because now $f_l f_f$ is in disjunction with some other terms (namely the ϵ's). To illustrate, consider conjunction $(f_l f_f \vee \epsilon)(f_l)(f_f) = (f_l f_f)(f_l)(f_f) \vee (\epsilon)(f_l)(f_f)$. Note that the second disjunct $(\epsilon)(f_l)(f_f)$ will identify the potential cross-matching $\{f_l, f_f\}$ anyway, regardless it might be a false-positive in the other term. Therefore, deleting $(f_l f_f)$ and rewriting $\mathcal{D}_e(\check{C}_1)$ as $\epsilon \vee \epsilon \vee \epsilon$ does not impact the safety test, unlike the previous case. Finally we can merge the *don't-care*'s, and thus $\mathcal{D}_e(\check{C}_1) = \epsilon$.

Figure 10 presents Procedure *EDNF*, which streamlines the computation of $\mathcal{D}_e(R)$ and $\mathcal{D}(R)$ for every subquery R in a tree. As just illustrated, we first compute the potential matchings M_p (line 1). Then, we structure the bottom-up process in a recursive subroutine *ednf*, which traverses the tree in post-order. At each node, step (1) of *ednf* first computes $\mathcal{D}(R)$ form $\mathcal{D}_e(\cdot)$ of the children. To obtain $\mathcal{D}_e(R)$, step (2) then simplifies $\mathcal{D}(R)$ by deleting useless terms (line 16–21) and merging extra ϵ's (line 22–23) with the rules that we intuitively illustrated.

We conclude our discussion with several important features of using EDNF. First, it allows us to focus on only the essential terms that may potentially contribute to cross-matchings. For example, when a query does not contain any constraint dependencies (in which case the potential matchings M_p consists of only single-constraint matchings), then all the EDNF will simply be ϵ. With this reduction, the safety check has virtually no cost. Furthermore, as we mentioned, we execute Procedure *EDNF* only once to prepare all the $\mathcal{D}(\cdot)$ and $\mathcal{D}_e(\cdot)$ required by the conjunctive nodes. Finally, we can reuse the potential matchings M_p computed in Procedure *EDNF* in the actual mapping process. For instance, in Example 6 we need to call *SCM*$(f_l f_f, K_{Amazon})$ to translate the simple conjunction $f_l f_f$. The algorithm will then match the constraints. For this task, we can simply reuse from M_p those relevant matchings, *i.e.*, $\{f_l\}$ and $\{f_l, f_f\}$. Finally, as we will formally show in Section 7.3, for our purpose of conjunct separation, EDNF is equivalent to full DNF.

7.2 Partitioning Conjunctive Queries

Based on the safety conditions, we next study how to safely partition conjuncts. This section presents Algorithm *PSafe*. As we explained in Section 6, Algorithm *PSafe* is the critical technique that our mapping algorithm (Algorithm *TDQM* in Figure 8) relies on for partitioning conjunctions.

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,

Procedure *EDNF***:** Computing EDNF (essential DNF) for Testing Separability
Input: • Q: an arbitrary query in the original context.
• K: the constraint mapping specification *w.r.t.* a target system T.
Purpose: computing $\mathcal{D}_e(\cdot)$ and $\mathcal{D}(\cdot)$ *w.r.t.* K for each node (*i.e.*, a subtree) of Q.
Procedure:
• $M_p \leftarrow \mathcal{M}(K, \mathcal{C}(Q))$ // *M_p is "global" to the recursive traversal of the whole tree.*
• *ednf*(Q) // *call* ednf *to compute $\mathcal{D}_e(\cdot)$ and $\mathcal{D}(\cdot)$ at each node from bottom up.*

Subroutine *ednf*(Q): // *recursively traverse Q in post-order to compute $\mathcal{D}_e(Q)$ and $\mathcal{D}(Q)$.*
(1) compute $\mathcal{D}(Q)$ from the EDNF of the children subqueries:
// **Case-1: disjunctive queries**, *i.e., at an $\vee$-node.*
• if $Q = \hat{D}_1 \vee \hat{D}_2 \vee \cdots \vee \hat{D}_n$:
– for each $\hat{D}_i$: $\mathcal{D}_e(\hat{D}_i) \leftarrow$ *ednf*$(\hat{D}_i)$ // *recursively call* ednf *for each disjunct.*
– $\mathcal{D}(Q) \leftarrow \mathcal{D}_e(\hat{D}_1) \vee \mathcal{D}_e(\hat{D}_2) \vee \cdots \vee \mathcal{D}_e(\hat{D}_n)$
// **Case-2: conjunctive queries**, *i.e., at an $\wedge$-node.*
Case-2: conjuntive queries
• else if $Q = \check{C}_1\check{C}_2 \cdots \check{C}_n$:
– for each $\check{C}_i$: $\mathcal{D}_e(\check{C}_i) \leftarrow$ *ednf*$(\check{C}_i)$ // *recursively call* ednf *for each conjunct.*
– $\mathcal{D}(Q) \leftarrow$ *Disjunctivize*$(\mathcal{D}_e(\check{C}_1)\mathcal{D}_e(\check{C}_2) \cdots \mathcal{D}_e(\check{C}_n))$ // *convert to disjunction.*
// **Case-3: single constraint**, *i.e., at a leaf node.*
• else if $Q = c$: $\mathcal{D}(Q) \leftarrow c$
(2) compute $\mathcal{D}_e(Q)$ by simplifying $\mathcal{D}(Q)$:
• $\mathcal{D}_e(Q) \leftarrow \mathcal{D}(Q)$ // *initialize $\mathcal{D}_e(Q)$ with $\mathcal{D}(Q)$.*
• for each disjunct $\hat{D}$ in $\mathcal{D}_e(Q)$:
– if ($\forall m \in M_p$ *s.t.* $m \cap \mathcal{C}(\hat{D}) \neq \phi$: // *any pontential matching m relevent to $\hat{D}$*
a. $m \subseteq \mathcal{C}(\hat{D})$, and // *$m$ is wholly contained in $\hat{D}$.*
b. either 1. m consist of a single constraint, or
2. $\exists \hat{D}'$ in $\mathcal{D}_e(Q)$ *s.t.* $m \cap \mathcal{C}(\hat{D}') = \phi$):
– replace $\hat{D}$ with ϵ // *nullify $\hat{D}$ as a "don't care"*
• simplify $\mathcal{D}_e(Q)$ with the following rules: // *to delete extra ϵ's.*
– $x\epsilon = x$, $x \vee x = x$, $xx = x$, for any x including $x = \epsilon$.
• return $\mathcal{D}_e(Q)$

Figure 10: Procedure *EDNF* for computing $\mathcal{D}(\cdot)$ and $\mathcal{D}_e(\cdot)$ of a query.

Algorithm *PSafe* can collect those indecomposable 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 11) 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 7.1.3). Recall that we compute the EDNF of conjuncts, and check if any cross-matchings exist across the combinations of the EDNF ingredients, as Example 11 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 12 illustrates this extension.

Example 12: We continue Example 11 to partition conjunction $\hat{Q}_{book}$. In Example 11, 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 12 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 decomposed. We next illustrate the idea. (Note that, to simplify presentation, in Example 13 we do not actually compute the conjunct EDNF.)

Example 13: 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 present Algorithm *PSafe* in Figure 11. This algorithm consists of two steps: (Note that, to begin with, the algorithm uses $\mathcal{D}(\hat{Q})$ and $\mathcal{D}_e(\check{C}_i)$'s computed form Procedure *EDNF* in Figure 10.) First, in step (1), for each cross-matching m, we find all the blocks (as subsets of conjuncts $\check{C}_i$) that minimally cover m. Finding such blocks is actually a *minimal cover* problem (formalized in line 9–10 of Figure 11). Note that in general m can be covered by multiple blocks, and not all of them will necessarily be included in the partition– We thus call them *candidate blocks*. In summary, step (1) finds all the cross-matchings as well as the candidate blocks that each covers one or more matchings. Next, in step (2), to form a partition, we choose some candidate blocks that minimally cover all the cross-matchings (line 16). We then ensure that the the chosen blocks are not overlapping, and that every conjunct is included in some block. To illustrate, we next show how Algorithm *PSafe* actually partitions the queries as intuitively discussed in Example 13.

Example 14 (Algorithm *PSafe*): Let us partition the queries $\hat{Q}_a$ and $\hat{Q}_b$ considered in Example 13. As illustrated in Figure 12, we first compute $\mathcal{D}(\cdot)$ and $\mathcal{D}_e(\cdot)$, and annotated the query trees (similarly to Figure 7). (Recall that we assume that the potential matchings for constraints x, y, u, v are $M_p = \{\{x, y\}, \{u\}, \{v\}\}$.)

First, consider $\hat{Q}_a = \check{C}_1\check{C}_2\check{C}_3 = (x)(y)(yu \vee v)$. As Figure 12 shows, $\mathcal{D}(\hat{Q}_a) = (x)(y)(y) \vee (x)(y)(\epsilon)$. (Note that by using EDNF we can focus on the dependent constraints x and y.) Term $(x)(y)(y)$ has a cross-matching $m_1 = \{x, y\}$. To find the candidate blocks for m_1, we look for some subsets of "ingredients" (x), (y), and (y) (from $\mathcal{D}_e(\check{C}_1)$, $\mathcal{D}_e(\check{C}_2)$ and $\mathcal{D}_e(\check{C}_3)$ respectively) that form minimal covers for $\{x, y\}$. In particular, $B_1 = \{\check{C}_1, \check{C}_2\}$ and $B_2 = \{\check{C}_1, \check{C}_3\}$ are such candidates. In addition, we also find a cross-matching

Algorithm *PSafe*: Partitioning a Conjunctive Query into Safe and Minimal Blocks
Input: • $\hat{Q}$: a conjunctive query in the original context; $\hat{Q} = \wedge(\{\check{C}_1, \check{C}_2, \ldots, \check{C}_n\})$.
• K: the constraint mapping specification *w.r.t.* a target system T.
Output: a partition of $\hat{Q}$ *w.r.t.* K.
Procedure:

// *This algorithm assumes that* $\mathcal{D}_e(\check{C}_i)$*'s and* $\mathcal{D}(\hat{Q})$ *have been computed with Procedure EDNF.*
// *Let* $\mathcal{D}_e(\check{C}_i) = \hat{I}_{i1} \vee \hat{I}_{i2} \vee \cdots \vee \hat{I}_{im_i}$, *and* $\mathcal{D}(\hat{Q}) = \vee(\{\hat{I}_{1k_1}\hat{I}_{2k_2} \cdots \hat{I}_{nk_n} | k_i \in [1:m_i]\})$.
(1) for each disjunct in $\mathcal{D}(\hat{Q})$, find any cross-matchings and the candidate blocks that cover them:
  • $M_\delta \leftarrow \phi$; $X \leftarrow \phi$ // M_δ*: to store cross-matchings;* X*: to store candidate blocks.* .
  • for each $\hat{D} = \hat{I}_{1k_1}\hat{I}_{2k_2} \cdots \hat{I}_{nk_n}$ in $\mathcal{D}(\hat{Q})$:
    – $\delta \leftarrow \mathcal{M}(\hat{D}, K) - \cup_{i=1}^{n} \mathcal{M}(\hat{I}_{ik_i}, K)$ // *the cross-matchings found in* $\hat{D}$.
    – for each m in δ:
      – add m to M_δ
      – find all subsets β of $\{1, 2, \ldots, n\}$, // $\mathcal{C}(\hat{I}_{ik_i})$ *is the set of constraints in* $\hat{I}_{ik_i}$.
        s.t.$\{\mathcal{C}(\hat{I}_{ik_i}) \mid i \in \beta\}$ is a minimal cover of m from $\{\mathcal{C}(\hat{I}_{1k_1}), \mathcal{C}(\hat{I}_{2k_2}), \ldots, \mathcal{C}(\hat{I}_{nk_n})\}$
      – for each such β: // β *represents a candidate block* B *that covers* m.
        – $B \leftarrow \{\check{C}_i \mid i \in \beta\}$ // B *is a block of conjuncts* $\check{C}_i$*'s that covers* m.
        – if ($B \notin X$): add B to X; $\tilde{B} \leftarrow \phi$ // *initialize a new candidate block* B.
        – add m to $\tilde{B}$ // $\tilde{B}$ *stores the matchings that* B *covers (can be more than one).*
(2) find a partition P *s.t.* each (cross-matching) m in M_δ is covered by some block in P:
  • find a subset P of X *s.t.*$\{\tilde{B} \mid B \in P\}$ is a minimal cover of M_δ from $\{\tilde{B} \mid B \in X\}$
  • merge any non-disjoint B_i and B_j in P // *make blocks disjoint.*
  • for each $\check{C}_i$ in $\{\check{C}_1, \check{C}_2, \ldots, \check{C}_n\}$: // *include any* $\check{C}_i$*'s that do not appear in any chosen block.*
    – if $\check{C}_i$ does not appear in any $B \in P$: add block $\{\check{C}_i\}$ to P
  • return P

Figure 11: Algorithm *PSafe* for partitioning conjunctive queries.

$m_2 = \{x, y\}$ for the second term $(x)(y)(\epsilon)$. (We view m_1 and m_2 as "distinct" since they appear in different terms.) In this case, m_2 can be covered by block B_1.

To determine a final partition, we want to select some minimal set of blocks from the candidates (*i.e.*, B_1 and B_2) to cover all the matchings (*i.e.*, m_1, and m_2). Intuitively, since m_2 is exclusively covered by B_1, we must include B_1. In fact, B_1 itself is sufficient since it actually covers both m_1 and m_2. Therefore we do not need to include B_2, and Algorithm *PSafe* outputs the partition $\{\{\check{C}_1, \check{C}_2\}, \{\check{C}_3\}\}$.

Let us next consider $\hat{Q}_b = \check{C}_1\check{C}_2\check{C}_3 = (x)(y \vee u)(y \vee v)$ also in Figure 12. Computed from $\check{C}_i$'s EDNF, $\mathcal{D}(\hat{Q}_b)$ has four terms $(x)(y)(y) \vee (x)(y)(\epsilon) \vee (x)(\epsilon)(y) \vee (x)(\epsilon)(\epsilon)$. Algorithm *PSafe* will check each term and find the cross-matchings $\{x, y\}$ in the first three. In particular, the cross-matching in $(x)(y)(\epsilon)$ can only be covered by block $B_1 = \{\check{C}_1, \check{C}_2\}$, while that in $(x)(\epsilon)(y)$ only by $B_2 = \{\check{C}_1, \check{C}_3\}$. Consequently, we need both B_1 and B_2 for the partition. Finally, merging the overlapping blocks, Algorithm *PSafe* outputs a 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 presented our algorithm for conjunction partitions. Note that this technique is essential to enable the effective handling of conjunctions. As Section 6 discussed, our translation mechanism (Algorithm *TDQM*) relies on Algorithm *PSafe* to partition conjuncts, and thus avoid the blind DNF conversions otherwise. Our discussion of Algorithm *PSafe* completes the overall translation framework. In the remaining of this section, we formally prove that Algorithm *PSafe* is correct.

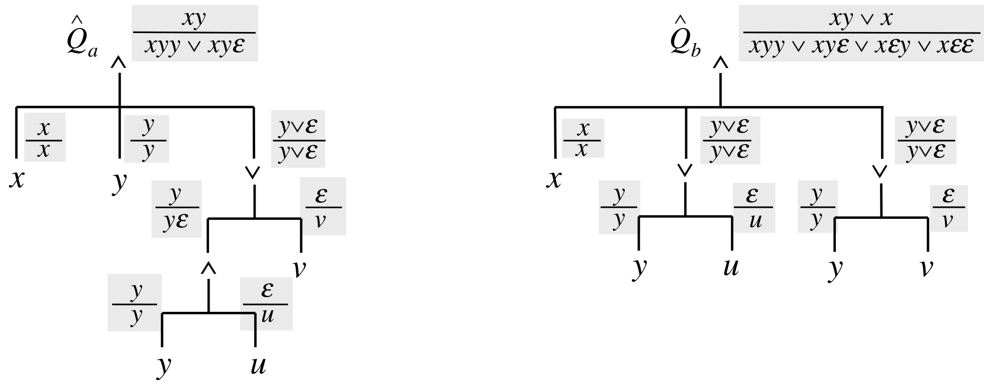

Figure 12: Query trees for Example 13.

7.3 Correctness of Algorithm *PSafe*

We have presented Algorithm *PSafe* for partitioning conjunctions; in this section we formally develop its correctness. Given $\hat{Q} = \check{C}_1 \cdots \check{C}_n$, Algorithm *PSafe* partitions the conjuncts $\check{C}_i$ into some blocks $B_1, \ldots, B_m$. Our goal is to show that the partition is safe, *i.e.*, $\mathcal{S}(\hat{Q}) = \mathcal{S}(\hat{B}_1) \cdots \mathcal{S}(\hat{B}_m)$. Furthermore, we will show that the partition is also minimal, *i.e.*, no blocks can be further partitioned safely. However, note that for the minimality we assume that Algorithm *PSafe* does not merge the overlapping blocks (*i.e.*, line 17 of Figure 11). (In other words, the minimality holds before block merging, if any.) The merging is to avoid repeated handling of conjuncts; a merged block can obviously be decomposed into some smaller (but overlapping) blocks.

This section develops the correctness as follows. First, Theorem 5 presents the basic property for a partition to be safe. Based on this property, Lemma 2 then shows that Algorithm *PSafe* is correct, modulo the use of EDNF (*i.e.*, if the algorithm used full DNF instead). Furthermore, in Lemma 3 we prove that using EDNF the algorithm gives exactly the same results as using full DNF. Finally, it follows from the lemmas that Algorithm *PSafe* does give safe and minimal partitions, which we state in Theorem 6.

We start with Theorem 5 to present the condition that characterizes a safe partition. Essentially, any safe partition must include some block to cover every cross-matching among the conjuncts. We will show that when and only when all the cross-matchings are covered can the partition be safe. Note that as a basis we focus on the cross-matchings identified among the disjuncts in the DNF of $\hat{C}_i$'s. We will discuss the equivalence of using EDNF in Lemma 3.

Theorem 5 (Safe Partition): Given a conjunctive query $\hat{Q} = \check{C}_1 \cdots \check{C}_n$, let the DNF of $\check{C}_i$ be $DNF(\check{C}_i) = \hat{I}_{i1} \vee \cdots \vee \hat{I}_{im_i}$ and that of $\hat{Q}$ be $DNF(\hat{Q}) = \sum_{k_i \in [1:m_i]} \hat{I}_{1k_1} \cdots \hat{I}_{nk_n}$. With respect to a mapping specification K, a partition P is safe if and only if for every disjunct $\hat{D} = \hat{I}_{1k_1} \cdots \hat{I}_{nk_n}$ in $DNF(\hat{Q})$ and for every cross-matching m in $\hat{D}$ (*i.e.*, $m \in \delta = \mathcal{M}(\hat{D}, K) - \cup_{i=1}^{n} \mathcal{M}(\hat{I}_{ik_i}, K)$), P contains some block that covers m. ∎

Proof: (if) Let the partition $P = \{B_1, \ldots, B_m\}$. Assuming that P covers every cross-matching, we show that the resulted conjunction $\hat{B} = \hat{B}_1 \cdots \hat{B}_m$ is safe according to Definition 6 (and thus P is safe). We can show the safety of $\hat{B}$ using the DNF-based scheme (*i.e.*, the "brute-force" approach discussed in Section 7.1.3). According to the scheme, we first compute $DNF(\hat{B})$. Note that $\hat{B} \equiv \hat{Q}$; we thus compute $DNF(\hat{B}) = DNF(\hat{Q}) = \sum_{k_i \in [1:m_i]} \hat{I}_{1k_1} \cdots \hat{I}_{nk_n}$.

To show that $\hat{B}$ is safe, we must in turn show the safety for each disjunct $\hat{D} = \hat{I}_{1k_1} \cdots \hat{I}_{nk_n}$ in $DNF(\hat{B})$. Since every $\hat{I}_{ik_i}$ is a simple conjunction, by Definition 5, this safety depends on if $\hat{D}$ has any matchings contributed by different blocks. (Note that now we are considering the safety in terms of the blocks B_j

rather than the individual conjuncts $\check{C}_i$.) To make explicit the contribution from each block, we group the ingredients $\hat{I}_{ik_i}$ according to their containing blocks. In other words, we rewrite $\hat{D}$ as $(\hat{X}_1)\cdots(\hat{X}_m)$, where each $\hat{X}_j$ represents the contribution from block B_j. Formally, $\hat{X}_j$ collects the term $\hat{I}_{ik_i}$ if the corresponding C_i participates in block B_j, *i.e.*,

$$\hat{X}_j = \prod_{C_i \in B_j} \hat{I}_{ik_i}. \tag{15}$$

To apply Definition 5, we must identify any cross-matchings in $\mathcal{M}(\hat{D}, K) - \cup_{j=1}^{m} \mathcal{M}(\hat{X}_j, K)$, denoted by δ_x. As we next explain, such cross-matchings cannot exist, *i.e.*, $\delta_x = \phi$, and thus $\hat{D}$ is safe (by Definition 5), which in turn leads to the safety of the conjunction $\hat{B}$. To see why, suppose that there exists a cross-matching m for $(\hat{X}_1)\cdots(\hat{X}_m)$. Since each X_j consists of several $\hat{I}_{ik_i}$ terms (as Eq. 15 shows), for m to come from some different $\hat{X}_j$'s, it is required that m come from different $\hat{I}_{ik_i}$'s. Thus m must also be a cross-matching for $(\hat{I}_{1k_1})\cdots(\hat{I}_{nk_n})$. Since by assumption each of such matchings is covered, the partition has some block B_c that covers m. Obviously, this covering ensures that m can be found within $\hat{X}_c$, *i.e.*, $m \in \mathcal{M}(\hat{X}_c, K)$. Consequently, $m \notin \mathcal{M}(\hat{D}, K) - \cup_{j=1}^{m} \mathcal{M}(\hat{X}_j, K)$. In other words, m cannot be a cross-matching for $(\hat{X}_1)\cdots(\hat{X}_m)$, a contradiction.

(only if) Suppose that there exists a cross-matching m for some disjunct $\hat{D} = (\hat{I}_{1k_1})\cdots(\hat{I}_{nk_n})$, such that no block in the partition covers m. As we discussed in the preceding part, in terms of the blocks, we write $\hat{D} = (\hat{X}_1)\cdots(\hat{X}_m)$. Note that, first, since m is a matching for $(\hat{I}_{1k_1})\cdots(\hat{I}_{nk_n})$, $m \in \mathcal{M}(\hat{D}, K)$. Second, since no block covers m, $m \notin \mathcal{M}(\hat{X}_j, K)$, for all $\hat{X}_j$. Therefore, we conclude that $m \in \mathcal{M}(\hat{D}, K) - \cup_{j=1}^{m} \mathcal{M}(\hat{X}_j, K)$. *i.e.*, m is a cross-matching for $(\hat{X}_1)\cdots(\hat{X}_m)$. In other words, the disjunct $\hat{D} = (\hat{X}_1)\cdots(\hat{X}_m)$ is not safe, and thus the partition P is not safe. ∎

We next show in Lemma 2 the correctness of Algorithm *PSafe*, modulo the use of EDNF. In other words, assuming that we simply use full DNF as EDNF, we show that the algorithm computes safe and minimal partitions. The safety follows directly from Theorem 5. As we explained, the minimality refers to the unmerged blocks (when there are overlapping blocks).

Lemma 2: Assume that we compute the EDNF of a query as its DNF. Given a conjunctive query $\hat{Q}$ and a mapping specification K, Algorithm *PSafe* partitions $\hat{Q}$ safely with respect to K. Without merging the overlapping blocks, the partition is also minimal. ∎

Proof: (safety) We first show that the partition of Algorithm *PSafe* is safe. Let the query be $\hat{Q} = \check{C}_1 \cdots \check{C}_n$. Note that when the EDNF of $\check{C}_i$ were computed as the full DNF, all $\mathcal{D}_e(\check{C}_i)$'s as well as $\mathcal{D}(\hat{Q})$ are simply the DNF of the corresponding queries (see line 1–2 of Figure 11). Observe that Algorithm *PSafe* first finds all the cross-matchings in any disjunct $\hat{D} = \hat{I}_{1k_1} \cdots \hat{I}_{nk_n}$ of $\mathcal{D}(\hat{Q})$ (step 1), and then chooses the blocks to cover all such matchings (step 2). It follows from Theorem 5 that the partition so formed is safe.

(minimality) We will prove the minimality by contradiction. Suppose that Algorithm *PSafe* returns the partition $P = \{B_1, \ldots, B_m\}$. Assume that some blocks can be further partitioned such that the resulted partition is still safe. Without loss of generality, let these non-minimal blocks be $B_1, \ldots, B_h$ $(h \le m)$. Suppose that we can partition each of these B_j's into some *proper* subsets $B_{j1}, \ldots, B_{jw_j}$ such that $B_{ji} \subset B_j$. In other words, the new partition is $P' = \{B_{11}, \ldots, B_{1w_1}, \ldots, B_{h1}, \ldots, B_{hw_h}, B_{h+1}, \ldots, B_m\}$.

We now show that this new partition P' cannot be safe. To illustrate let us consider B_1 (among the non-minimal blocks). Referring to Figure 11, note that Algorithm *PSafe* (in step 2) finds P as a minimal cover of M_δ (which collects all the cross-matchings). Since P is minimal, B_1 cannot be redundant for the covering. In other words, there exists some matching $m \in M_\delta$, such that m is *exclusively* covered by B_1. Otherwise, if every matching in M_δ is covered by some other blocks, then P is not minimal because $(P - \{B_1\})$ still covers M_δ.

However, this cross-matching m cannot be covered by any blocks in P', and therefore by Theorem 5 the new partition is not safe. Since Algorithm *PSafe* ensures (in step 1) that B_1 is a minimal cover for m, none of the proper subsets $B_{11}, \ldots, B_{1w_1}$ can cover m. Meanwhile, as m is exclusively covered by B_1, no other blocks $B_2, \ldots, B_m$ or their subsets B_{ji} can cover m. Overall, m is not covered in the new partition. By Theorem 5, the new partition is not safe, a contradiction. ■

To complete our discussion for the correctness of Algorithm *PSafe*, we must show that using EDNF (essential DNF) gives the same results as that of using full DNF. That is, for the purpose of Algorithm *PSafe*, it is equivalent to use either EDNF or DNF. Lemma 3 presents this result.

Lemma 3: For any conjunctive query $\hat{Q} = \check{C}_1 \cdots \check{C}_n$, Algorithm *PSafe* gives the same result regardless of whether we use the EDNF or the full DNF of $\check{C}_i$ as $\mathcal{D}_e(\check{C}_i)$'s to compute $\mathcal{D}(\hat{Q})$. ■

Proof: Essentially, Algorithm *PSafe* identifies (in step 1) and covers (in step 2) the cross-matchings in $\mathcal{D}(\hat{Q})$. Consequently, the difference between using EDNF and DNF for computing $\mathcal{D}(\hat{Q})$ is in the resulting cross-matchings, which is designated as M_δ in Figure 11. To show the equivalence, suppose that using full DNF will result in the cross-matching set M_d while using EDNF will result in M_e instead. (Note that we use the different subscripts to distinguish the variables in the DNF and EDNF-based schemes.) Our proof consists of two parts:

(a) M_d and M_e are *equivalent* in the sense that $\forall m_d \in M_d$, $\exists m_e \in M_e$ (and vice versa), such that m_d and m_e have exactly the same set of candidate blocks.

(b) For such equivalent M_d and M_e, a partition P (as a set of some candidate blocks) is a minimal cover for M_d if and only if it is a minimal cover for M_e.

Intuitively, part (a) and (b) correspond to step 1 and 2 in Algorithm *PSafe*. In other words, in part (a) we show that M_d and M_e are equivalent for our purpose. Note that when M_d and M_e are equivalent, they will be associated with the same set of candidate blocks (*i.e.*, the variable X in Algorithm *PSafe*). Then, part (b) further shows that such equivalent cross-matching sets requires exactly the same minimal covers from the candidate blocks. Overall, we can conclude that Algorithm *PSafe* results in the same partitions regardless of using essential or full DNF, and thus the equivalence is proven.

(a) In this part, we will show that M_d and M_e are equivalent. Note that the EDNF of a query is essentially a transformation of the full DNF, as computed with Procedure *EDNF*. These transformations are actually done in step 2 of the procedure (Figure 10). Specifically, the transformations apply (1) the *nullifying* rules which nullify the useless terms (line 17-22, Figure 10) and (2) the *simplifying* rules which simplify the expression by deleting the nullifying symbol ϵ and merging repeating terms (line 23-24, Figure 10). Observe that without these transformations, the procedure would have resulted in the DNF instead of the EDNF of a query. Thus to show the equivalence of M_d and M_e, we must show that the equivalence holds under either transformation, which we discuss in turn next.

(1) **(nullifying rules)** We first focus on the nullifying rules (and for now omit the simplifying rules in Procedure *EDNF*). The EDNF so computed for disjunct $\check{C}_i$ has the form

$$\mathcal{D}_e^*(\check{C}_i) = \hat{I}_{i1}\boldsymbol{\epsilon}_{i1} \vee \cdots \vee \hat{I}_{im_i}\boldsymbol{\epsilon}_{im_i}. \tag{16}$$

(We write here as $\mathcal{D}_e^*(\check{C}_i)$ and reserve $\mathcal{D}_e(\check{C}_i)$ for the actual EDNF with the simplifying rules also applied.) Here every $\boldsymbol{\epsilon}_{ij}$ represents some zero or more nullifying symbols ϵ, each of which replaces a useless intermediate term (designated as disjunct $\hat{D}$ in line 17-22 of Figure 10). In fact, if these useless terms were not nullified, Eq. 16 would become the full DNF. In other words, assuming that $\mathbf{U}_{ij}$ consists of those (zero or more) useless terms that $\boldsymbol{\epsilon}_{ij}$ replaces, the corresponding DNF can be written as

$$DNF(\check{C}_i) = \hat{I}_{i1}\mathbf{U}_{i1} \vee \cdots \vee \hat{I}_{im_i}\mathbf{U}_{im_i}. \tag{17}$$

• First, we show that for every $m_d \in M_d$ (in the DNF-based scheme), there exists $m_e \in M_e$ (in the EDNF-based scheme), such that m_d and m_e have the same candidate blocks. (The converse will be shown later.) Note that in the DNF-based scheme, with $\mathcal{D}(\hat{Q})$ computed from $DNF(\check{C}_i)$ (as given in Eq. 17), every such m_d is a cross-matching for some conjunction (as a disjunct in $\mathcal{D}(\hat{Q})$) $\hat{D}_d = (\hat{I}_{1k_1}\mathbf{U}_{1k_1})\cdots(\hat{I}_{nk_n}\mathbf{U}_{nk_n})$. We will show that the same matching (*i.e.*, $m_e = m_d$) exists in the corresponding term (of the EDNF-based scheme) $\hat{D}_e = (\hat{I}_{1k_1}\boldsymbol{\epsilon}_{1k_1})\cdots(\hat{I}_{nk_n}\boldsymbol{\epsilon}_{nk_n})$.

To begin with, we show that every $\mathbf{U}_{ik_i}$ is indeed *irrelevant* to m_d, *i.e.*, $m_d \cap \mathcal{C}(\mathbf{U}_{ik_i}) = \phi$. (Recall that $\mathcal{C}(Q)$ denotes the set of constraints in query Q.) As a cross-matching in $\hat{D}_d$, m_d cannot be contained in any $\hat{I}_{ik_i}\mathbf{U}_{ik_i}$, for all i, *i.e.*,

$$m_d \not\subseteq \mathcal{C}(\hat{I}_{ik_i}\mathbf{U}_{ik_i}), \forall i \tag{18}$$

Furthermore, by definition each $\mathbf{U}_{ik_i}$ consists of some useless terms U. The nullifying rules (line 17-22, Figure 10) require that either $m_d \cap \mathcal{C}(U) = \phi$ or $m_d \subseteq \mathcal{C}(U)$. Since the latter contradicts Eq. 18, the former must hold for every U, and thus $m_d \cap \mathcal{C}(\mathbf{U}_{ik_i}) = \phi$.

Corresponding to m_d, we can find some $m_e \in M_e$ as required. Obviously, for every $\hat{D}_d = (\hat{I}_{1k_1}\mathbf{U}_{1k_1})$ $\cdots(\hat{I}_{nk_n}\mathbf{U}_{nk_n})$, the EDNF-based scheme has a counterpart $\hat{D}_e = (\hat{I}_{1k_1}\boldsymbol{\epsilon}_{1k_1})\cdots(\hat{I}_{nk_n}\boldsymbol{\epsilon}_{nk_n})$. As just explained, since $m_d \cap \mathcal{C}(\mathbf{U}_{ik_i}) = \phi$, $\mathbf{U}_{ik_i}$ is indeed irrelevant for covering m_d. In other words, for the purpose of covering m_d, each term $\hat{I}_{ik_i}\mathbf{U}_{ik_i}$ is equivalent to $\hat{I}_{ik_i}\boldsymbol{\epsilon}_{ik_i}$. More formally, let's consider m_e $= m_d$. Given that m_d is a cross-matching for $\hat{D}_d$, we can find m_e as a cross-matching for $\hat{D}_e$, because $m_d \cap \mathcal{C}(\hat{I}_{ik_i}\mathbf{U}_{ik_i}) = m_e \cap \mathcal{C}(\hat{I}_{ik_i}\boldsymbol{\epsilon}_{ik_i}) = m_d \cap \mathcal{C}(\hat{I}_{ik_i})$. It also follows that m_e will be covered by the same candidate blocks as that of m_d.

• Next, we show the converse; *i.e.*, for every $m_e \in M_e$, there exists $m_d \in M_d$, such that m_d and m_e have the same candidate blocks. Since the EDNF-based scheme uses $\mathcal{D}_e^*(\check{C}_i)$'s (as given in Eq. 16), m_e is a cross-matching found in some conjunction $\hat{D}_e = (\hat{I}_{1k_1}\boldsymbol{\epsilon}_{1k_1})\cdots(\hat{I}_{nk_n}\boldsymbol{\epsilon}_{nk_n})$. Our proof will construct a counterpart $\hat{D}_d$ (for the DNF-based scheme) containing the desired cross-matching m_d corresponding to m_e.

Essentially, the construction is simply to "recover" the nullified terms $\boldsymbol{\epsilon}_{ij}$. Recall that each $\boldsymbol{\epsilon}_{ij}$ consists of zero or more ϵ's representing some useless terms U. The nullifying rules (line 17-22, Figure 10) ensure that, with respect to m_e, every useless term U satisfies either condition (1) $m_e \cap \mathcal{C}(U) = \phi$, or (2) $m_e \subseteq \mathcal{C}(U)$. In the latter case, there must exist some other term U' that parallels U in the same subquery such that $m_e \cap \mathcal{C}(U') = \phi$ (line 21, Figure 10).

We now construct $\hat{D}_d$. Starting from $\hat{D}_e$, we recover each ϵ in every $(\hat{I}_{ik_i}\boldsymbol{\epsilon}_{ik_i})$ as follows: If the useless term U behind ϵ satisfies condition (1), we simply recover the ϵ to U (note that $m_e \cap \mathcal{C}(U) = \phi$). Else, when U satisfies condition (2), we instead recover the ϵ to the corresponding U' (note that $m_e \cap \mathcal{C}(U') = \phi$). Supposing that $\mathbf{U}_{ik_i}$ designates the conjunction of those recovered terms, we have recovered $(\hat{I}_{ik_i}\boldsymbol{\epsilon}_{ik_i})$ to $(\hat{I}_{ik_i}\mathbf{U}_{ik_i})$. Consequently we construct $\hat{D}_d$ as $(\hat{I}_{1k_1}\mathbf{U}_{1k_1})\cdots(\hat{I}_{nk_n}\mathbf{U}_{nk_n})$. Note that this construction ensures that $m_e \cap \mathbf{U}_{ik_i} = \phi$.

We can find the desired matching m_d in $\hat{D}_d$ just constructed. First, with all the nullified terms recovered, $\hat{D}_d = (\hat{I}_{1k_1}\mathbf{U}_{1k_1})\cdots(\hat{I}_{nk_n}\mathbf{U}_{nk_n})$ is the DNF-based counterpart of $\hat{D}_e = (\hat{I}_{1k_1}\boldsymbol{\epsilon}_{1k_1})\cdots(\hat{I}_{nk_n}\boldsymbol{\epsilon}_{nk_n})$. Furthermore, as just explained our construction ensures that $m_e \cap \mathbf{U}_{ik_i} = \phi$. Therefore, supposing that $m_d = m_e$, we again obtain that $m_d \cap \mathcal{C}(\hat{I}_{ik_i}\mathbf{U}_{ik_i}) = m_e \cap \mathcal{C}(\hat{I}_{ik_i}\boldsymbol{\epsilon}_{ik_i}) = m_d \cap \mathcal{C}(\hat{I}_{ik_i})$. Therefore, the DNF-based scheme will identify m_d as a cross-matching (*i.e.*, $m_d \in M_d$) in $\hat{D}_d$, and m_d and m_e will require the same candidate blocks.

(2) **(simplifying rules)** Suppose that we further apply the simplifying rules (line 23-24, Figure 10) to $\mathcal{D}_e^*(\hat{C}_i)$ (given in Eq. 16), resulting in the actual EDNF denoted by $\mathcal{D}_e(\hat{C}_i)$. We next show that the simplifying rules will not alter the results obtained with $\mathcal{D}_e^*(\hat{C}_i)$. Since $\mathcal{D}_e^*(\hat{C}_i)$ and $DNF(\hat{C}_i)$ are equivalent under Algorithm *PSafe* (as just proven), the equivalence of $\mathcal{D}_e(\hat{C}_i)$ and $DNF(\hat{C}_i)$ follows by transitivity.

To begin with, the equivalence is obviously preserved by the rules $x \vee x = x$ and $xx = x$ since either side of the equations are logically equivalent. It is straightforward to show that this well-known Boolean *idempotency* applies to our partition algorithm, and we thus omit the details here.

Furthermore, the deletion of ϵ's with the rule $x\epsilon = x$ will also preserve the equivalence. Deleting the ϵ's from Eq. 16 results in $\mathcal{D}_e(\check{C}_i) = \hat{I}_{i1} \vee \cdots \vee \hat{I}_{im_i}$. Since by definition $\mathcal{C}(\boldsymbol{\epsilon}_{ij}) = \phi$, every $\boldsymbol{\epsilon}_{ij}$ will not contribute to any matchings. Therefore, it is clear that the equivalence is also preserved through the simplifying rules.

(b) We show that if a partition P is a minimal cover for M_d, then it also minimally covers M_e. The converse follows immediately since M_d and M_e are symmetrical.

First, as a cover of M_d, P must also cover M_e; *i.e.*, any matching m_e in M_e is covered by some block in P. To see why, note that according to (a), there exists a matching m_d in M_d, such that m_d and m_e has the same candidate blocks. Because P covers M_d, P includes some block B that is among the candidate blocks of m_d. Consequently, B is also a candidate block of m_e (and thus B covers m_e). Therefore P is also a cover for M_e.

Furthermore, P is minimal (as a cover) for M_e. Otherwise, suppose that some proper subset P' of P also covers M_e. With the converse of the preceding reasoning, P' is also a cover for M_d, which is a contradiction to the assumption that P is minimal for M_d. ■

Finally, putting together Lemma 2 and 3, we obtain immediately that Algorithm *PSafe* indeed partitions conjunctive queries safely and minimally. We therefore conclude our discussion of correctness with Theorem 6.

Theorem 6 (Correctness of Algorithm *PSafe*): Given a conjunctive query $\hat{Q}$ and a mapping specification K, Algorithm *PSafe* partitions $\hat{Q}$ safely with respect to K. Without merging the overlapping blocks, the partition is also minimal. ■

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; specifically, the partitioning of conjuncts respects constraint dependencies. In the preceeding sections we have given the formal proof for the correctness of the algorithms. In particulr, Theorem 2 shows that Algorithm *TDQM* does generate the minimal subsuming mappings.

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 [22] 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 7.1.3 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 EDNF). 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 EDNF will pay a cost "proportional" to the degree of dependency (informally speaking). Recall that we use EDNF that eliminates useless terms (Example 11). In other words, $\check{C}_i$'s EDNF 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 EDNF 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 EDNF's of $\check{C}_i$'s are simply ϵ (*e.g.*, $\mathcal{D}_e(\check{C}_1) = \epsilon$ in Example 11). 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, 20] 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] Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):51–60, March 1992.

[2] Jeffrey D. Ullman. Information integration using logical views. In *Proceedings of the 6th International Conference on Database Theory*, Delphi, Greece, January 1997. Springer, Berlin.

[3] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the 22nd VLDB Conference*, pages 251–262, Bombay, India, 1996. VLDB Endowment, Saratoga, Calif.

[4] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Query-answering algorithms for information agents. In *Proceedings of the 13th National Conference on Artificial Intelligence, AAAI-96*, Portland, Oreg., August 1996. AAAI Press, Menlo Park, Calif.

[5] Yannis Papakonstantinou, Héctor García-Molina, and Jeffrey Ullman. Medmaker: A mediation system based on declarative specifications. In *Proceedings of the 12th International Conference on Data Engineering*, New Orleans, La., 1996.

[6] Yannis Papakonstantinou, Héctor García-Molina, Ashish Gupta, and Jeffrey Ullman. A query translation scheme for rapid implementation of wrappers. In *Proceedings of the 4th International Conference on Deductive and Object-Oriented Databases*, pages 161–186, Singapore, December 1995. Springer, Berlin.

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

[8] Michael R. Genesereth, Arthur M. Keller, and Oliver M. Duschka. Infomaster: An information integration system. In *Proceedings of the 1997 ACM SIGMOD Conference*, Tucson, Ariz., 1997. ACM Press, New York.

[9] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang. Optimizing queries across diverse data sources. In *Proceedings of the 23rd VLDB Conference*, pages 276–285, Athens, Greece, August 1997. VLDB Endowment, Saratoga, Calif.

[10] Mary Tork Roth and Peter M. Schwarz. Don't scrap it, wrap it! a wrapper architecture for legacy data sources. In *Proceedings of the 23rd VLDB Conference*, pages 266–275, Athens, Greece, August 1997. VLDB Endowment, Saratoga, Calif.

[11] Olga Kapitskaia, Anthony Tomasic, and Patrick Valduriez. Dealing with discrepancies in wrapper functionality. Technical Report RR-3138, INRIA, 1997.

[12] Héctor García-Molina, Wilburt Labio, and Ramana Yerneni. Capability sensitive query processing on internet sources. In *Proceedings of the 15th International Conference on Data Engineering*, Sydney, Australia, March 1999. Accessible at `http://www-db.stanford.edu/`.

[13] Anand Rajaraman, Yehoshua Sagiv, and Jeffrey D. Ullman. Answering queries using templates with binding patterns. In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 105–112, San Jose, Calif., May 1995.

[14] Alon Y. Levy, Anand Rajaraman, and Jeffrey D. Ullman. Answering queries using limited external query processors. In *Proceedings of the 15th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 27–37, Montreal, Canada, June 1996.

[15] Chen-Chuan K. Chang, Héctor García-Molina, and Andreas Paepcke. Boolean query mapping across heterogeneous information sources. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):515–521, August 1996.

[16] Chen-Chuan K. Chang, Héctor García-Molina, and Andreas Paepcke. Boolean query mapping across heterogeneous information sources (extended version). Technical Report SIDL-WP-1996-0044, Stanford Univ., September 1996. Accessible at `http://www-diglib.stanford.edu`.

[17] Chen-Chuan K. Chang and Héctor García-Molina. Conjunctive constraint mapping for data translation. In *Proceedings of the Third ACM International Conference on Digital Libraries*, Pittsburgh, Pa., June 1998. ACM Press, New York.

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

[19] Yannis Papakonstantinou, Ashish Gupta, and Laura Haas. Capabilities-based query rewriting in mediator systems. In *Proceedings of the 4th International Conference on Parallel and Distributed Information Systems (PDIS 1996)*, Miami Beach, Flor., 1996.

[20] Chen-Chuan K. Chang, Héctor García-Molina, and Andreas Paepcke. Predicate rewriting for translating boolean queries in a heterogeneous information system. *ACM Transactions on Information Systems*, 17(1):1–39, January 1999.

[21] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, Reading, Mass., 1974.

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