Mind Your Vocabulary:
Query Mapping Across Heterogeneous Information Sources

Chen-Chuan K. Chang
Electrical Engineering Department
Stanford University
Stanford, CA 94305-9040, USA
{kevin,hector}@db.stanford.edu

H´ector Garc´ıa-Molina
Computer Science Department

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 supported, our framework explores relaxations (semantic rewritings) into the closest supported version. Our most sophisticated supported, our framework explores relaxations (semantic rewritings) into the closest supported version. Our most sophisticated supported, our framework explores relaxations (semantic rewritings) into the closest supported version. Our most sophisticated supported, our framework explores relaxations (semantic rewritings) into the closest supported version. Our most sophisticated supported, our framework explores relaxations (semantic rewritings) into the closest supported version.

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.aces.com and shopping.yahoo.com). In particular, the mediator exports an integrated view Book(title, ln, fn, ...) with attributes for title, author last name, first name, etc. To search for books, users specify constraints in their queries. Suppose that a user is looking for books by Tom Clancy, i.e., the constraint query \( Q \) is \([\text{fn} = "\text{Tom}"] \land [\text{ln} = "\text{Clancy}"]\).

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 \([\text{author} = "\text{Clancy},\text{Tom}"]\).

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 \( Q_c = [\text{author contains Tom}] \land [\text{author contains Clancy}] \). 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 \([\text{attr1 op value}]\) or \([\text{attr1 op attr2}]\). 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 \([\text{car-type} = "\text{ford}-\text{taurus}"] \land [\text{year} = 1994]\) may yield \([\text{make} = "\text{ford}"] \land [\text{model} =...\]
impose constraints on the first name alone. Thus simply "Clancy" to "Klancy", we may obtain only a suboptimal mapping. To illustrate, for a constraint 

\[ f_1 = \text{fn} = \text{"Clancy"}, \quad f_2 = \text{ln} = \text{"Klancy"}, \quad f_3 = \text{fn} = \text{"Tom"}. \]

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 \( S(X) \) denote the mapping of query \( X \). Separating \( C_1 \) and \( C_2 \) (as well as \( f_1 \) and \( f_2 \)), we obtain the mapping \( Q_a = S(C_1) \land S(C_2) = [S(f_1) \lor S(f_2)] \land S(f_3) \). Note that \( S(f_3) = \text{True} \) (i.e., no constraint) because Amazon cannot impose constraints on the first name alone. Thus \( Q_a = S(f_1) \lor S(f_2) = [\text{author} = \text{"Clancy"}] \lor [\text{author} = \text{"Klancy"}] \).

\( Q_a \) is actually not “minimal”; it is not as selective as \( Q_b = [\text{author} = \text{"Clancy",Tom"}] \lor [\text{author} = \text{"Klancy",Tom"}] \) (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 \([\text{ln} = \text{L}]\) can be mapped to \([\text{author} = \text{A}]\), 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 \([\text{fn} = \text{F}]\) and \([\text{ln} = \text{L}]\) can be mapped to \([\text{author} = \text{A}]\), 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 \((f_1 \lor f_2) \land f_3 \land f_4\). We may have a rule for mapping \((f_3 \land f_4)\) and another for \((f_2 \land f_3)\). 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 \((f_1 \land f_4)\) and \((f_3, f_4)\) 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, \textit{i.e.}, the query with the smallest parse tree, out of the possible translations. The algorithm does not blindly convert queries to DNF, which would be easier to translate, but expensive. Instead it performs a top-down mapping of a query tree, and does local query structure conversion only when necessary.

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

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

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

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

## 2 The Constraint Mapping Problem

We describe the constraint mapping problem in a common mediation architecture \([1, 2]\) for integrating heterogeneous sources. In such systems, wrappers unify the source data models, and mediators interact with the wrappers to process queries transparently. A mediator exports integrated mediator views for users to formulate queries. Thus, a user query \( U \) over some views \( V_1, \ldots, V_h \) where \( C \), or algebraically \( U = \sigma_C(V_1 \times \cdots \times V_h) \), 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 \([V_i, \text{attr1} \text{ op value}],\) or a join condition \([V_i, \text{attr1} \text{ op } V_j, \text{attr2}],\) where \text{attr1} and \text{attr2} are attributes of view \(V_i\) and \(V_j\) respectively.

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

For source execution, the mediator must rewrite a user query in terms of the source relations. Thus, with view expansion, \(U\) will be rewritten to the following form in Eq. 1, where \(R_i\) is the cross-product of all the source relation instances that a particular source \(T_i\) contributes to any viewed queries, and \(X\) 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.

\[
U = \sigma_Q(R_1 \times \cdots \times R_n \times X)
\]

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 \(X\) can present new attributes \((\text{e.g.,} \, \text{ln} \, \text{and} \, \text{fn} \, \text{that replace} \, \text{author})\) 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 \(S_i(Q)\), to retrieve the relevant subset of \(R_i\). 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.,

\[
U = \sigma_F(\sigma_{S_1(Q)}(R_1) \times \cdots \times \sigma_{S_n(Q)}(R_n) \times X).
\]

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

\[
Q = F \land S_1(Q) \land \cdots \land S_n(Q).
\]

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

Suppose that a user is looking for the papers written by some CS faculty interested in data mining. The constraint query is \(Q = a:\text{fac}.ln = \text{pub}.ln \land b:\text{fac}.fn = \text{pub}.fn \land c:\text{fac}.bib \text{contains data(near)}\text{mining} \land d:\text{fac}.dept = \text{cs}\).

Let’s first consider the mapping for source \(T_1\), i.e., for relations \(A\) and \(B\). The join conditions \(a \land b\) together map to \(x_1 : [A.au = B.name]\). If source \(T_1\) does not support the proximity operator near, rather than dropping constraint \(c\), we can relax it to \(x_2:[B.bib \text{contains data}]\land x_3:[B.bib \text{contains mining}]\) that requires only the occurrences of keywords. Lastly, constraint \(d\) maps to \text{True} (it can only be processed in \(T_2\)). Thus, \(S_1(Q) = x_1 \land x_2 \land x_3\).

We next perform the mapping for source \(T_2\), which contributes relation \(C\). All the constraints except \(d\) map to \text{True}. Suppose that \(T_2\) uses department code 230 for CS, thus \(S_2(Q) = [C.dept = 230]\).

Finally, the filter query \(F\) is simply the constraint \(c\) \((\text{i.e.,} \, F = c)\), the only constraint that is not fully realized at the underlying sources. Thus, \(Q = F \land S_1(Q) \land S_2(Q)\).

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 \(S_u(Q)\): To begin with, \(S_u(Q)\) must be expressible in target \(T_u\); i.e., \(S_u(Q)\) contains only those constraints that \(T_u\) supports with its schema and capability. (Thus, \(S_u(Q)\) uses only the native vocabulary of \(T_u\).)

Furthermore, \(S_u(Q)\) logically subsumes \(Q\); note that we can rewrite Eq. 3 as \(Q = F_u \land S_u(Q)\) (where \(F_u\) is the conjunction of \(F\) and \(S_u(Q)\), \(i \neq u\)). For a relation \(D\) in this case \(D = R_1 \times \cdots \times R_n \times X)\), \(Q\) subsumes \(Q\) if \(\sigma_Q(D) \supset \sigma_Q(D)\) 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 \(S_u(Q)\) on the \(R_u\) 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 \(S_u(Q)\) to return as few extra tuples as possible; i.e., \(S_u(Q)\) should be the most selective mapping. In this case we say that \(S_u(Q)\) 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 \(S_u(Q)\) is the minimal subsuming mapping of a constraint query \(Q\) w.r.t. the target context \(T_u\), if (1) \(S_u(Q)\) is expressible in \(T_u\), (2) \(S_u(Q)\) subsumes \(Q\), and (3) \(S_u(Q)\) is minimal, i.e., there is no query \(Q’\) such that (i) \(Q’\) satisfies 1 and 2, and (ii) \(S_u(Q)\) 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 $U$). Also, we write the mapping as $S(Q)$ (without a subscript) when the target source is clear as in Example 2. In addition, due to space limitations, we only consider selection constraints (of the form $[\text{attr1 op value}]$) in this paper. Our framework can handle join constraints (of the form $[\text{attr1 op attr2}]$) as well. We discuss the extensions in an extended technical report [17].

3 Related Work

While information integration has long been an active research area [1, 2, 18], the constraint mapping problem we study in this paper has not been addressed thoroughly. 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 constraints. However, these capability-description frameworks focus on the grammatical 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 $[\text{attr} = \text{value}]$, 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 $[\text{ln} = \text{"Clancy"}]$ and $[\text{fn} = \text{"Tom"}]$ to $[\text{author} = \text{"Clancy,Tom"}]$). 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 $\text{LnFnName}$ that is an inverse of the conversion function $\text{NameLnFn}$ 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 "Clancy"])). Moreover, constraint mapping must also map operators to respect source capabilities. For instance, in Example 3, the mapping from data(near)mining to data(\text{\textbackslash})mining has nothing to do with data conversion.

- We perform systematic semantic mapping of constraints (with human-specified rules); most other systems do not take advantage of relaxing an unsupported constraint semantically. The wrappers of these systems simply translate a constraint syntactically (e.g., from $[\text{ln} = \text{"Clancy"}]$ to the native command "lookup -ln \text{"Clancy"}") 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 \text{\textbackslash} in Example 3).

- We efficiently process complex queries. Most other systems focus on simple conjunctive queries, or process complex queries in DNF, which is expensive in general. In contrast, our algorithms do not assume DNF (Section 6).

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

4 Simple-Conjunction Queries

Query translation must rely on human expertise. In this section we present a rule-based scheme that codifies such expertise. The scheme relies on rules to indicate what groups of constraints need to be mapped as a unit, and what user-provided functions must be executed to actually transform values (e.g., to change the units or encoding of values). As we will see, the human-provided rules only
specify how to translate the smallest grouping of basic constraints, e.g., a pair of constraints that must be considered together for proper translation. The translation of full queries is then performed by a query translation algorithm, which relies on the rules to transform the basic constraints involved. In this section we describe the basic translation rules, and we discuss an algorithm that can translate any simple conjunctive query. In later sections we then present algorithms that can handle general Boolean queries. Our rule specifications are based on rules we developed earlier for data translation [20]. Here we adapt this framework for query translation. Please refer to [20] for a more formal definition of the rule framework.

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

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

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

---

\(^1\)We assume a target context based on the “power search” interface at www.amazon.com, with slight changes for the purpose of illustration.

---

\(Q_1 = f_1 \land f_2 \land f_3 \land f_4 \land f_5\) \(f_1: [t] = "Smith"\)
\(f_2: [a1] = "\text{java(near)jdk}\"
\(f_3: [\text{year}] = 1997\)
\(f_4: [\text{month}] = 5\)
\(f_5: [\text{contains}] = \text{www}\)

\(S_1 = a_a \land a_{a1} \land a_{a2} \land a_{a3}\)
\(a_a: [\text{author}] = "\text{Smith}\"
\(a_{a1}: [\text{ti-word contains}] = \text{java(\&)jdk}\"
\(a_{a2}: [\text{subject-word contains}] = \text{www}\"
\(a_{a3}: [\text{isbn}] = "081815181Y\"

\(Q_2 = f_6 \land f_7 \land f_8 \land f_9\)
\(f_6: \text{publisher} = \text{"oreilly"}\)
\(f_7: [\text{ti}] = \text{"jdk for java"}\)
\(f_8: [\text{category}] = \text{"D.3"}\)
\(f_9: [\text{id-no}] = \text{"081815181Y"}\)

\(S_2 = a_y \land a_{a6} \land a_{a7} \land a_{a8}\)
\(a_y: [\text{publisher}] = \text{"oreilly"}\"
\(a_{a6}: [\text{title starts}] = \text{"jdk for java"}\"
\(a_{a7}: [\text{subject}] = \text{"programming"}\"
\(a_{a8}: [\text{isbn}] = "081815181Y\"

Figure 2: Mapping simple-conjunction queries.

For example, rule \( R_4 \) in \( K_{Amazon} \) (Figure 3) maps a contains constraint on ti to one on attribute ti-word (e.g., \( f_1 \) to \( a_{a1} \) in Figure 2). When pattern [ti contains \( P_1 \)] matches a constraint (e.g., \( f_1 \)), the variable \( P_1 \) (in capitalized symbol) is bound to the corresponding constant, i.e., \( P_1 = \text{"java(near)jdk"}\). The matching of the head will fire the actions in the tail. In particular, it calls upon function \( \text{RewriteTextPat} \) to rewrite the text pattern \( P_1 \). As Amazon does not support near, \( P_1 \) is rewritten to \( P_2 = \text{"java(\&)jdk"}\). (For instance, reference [21] describes a general procedure for translating such IR predicates.) As we mentioned, the functions (as well as the conditions in the head) are supplied externally, and in principle can be written in any programming languages. Finally, the emission (i.e., the emit clause) of the rule outputs the mapping as [ti-word contains \( P_2 \) i.e., \( a_{a1} \) in Figure 2].

A rule can use conditions (i.e., predicate functions) to restrict the matchings. For instance, while the pattern in \( R_1 \) can match any constraint, condition \( \text{SimpleMapping}(A1) \) restricts the matchings. We denote the set of all the mappings of a rule \( R \) for query \( \hat{Q} \) by \( M(\hat{Q}, R) \). For instance, consider \( R_1 \) and assume that \( \text{SimpleMapping}(A1) \) holds only for attributes id-no and publisher. Referring to Figure 2, we get two matchings for \( \hat{Q}_2 \) (i.e., \( M(\hat{Q}_2, R_1) = \{\{f_y, f_z\}\}\)) but none for \( \hat{Q}_1 \) (i.e., \( M(\hat{Q}_1, R_1) = \{\}\)). Moreover, a matching can have multiple constraints. For example, constraints \( f_y \) and \( f_m \) in \( \hat{Q}_1 \) together match \( R_4 \), i.e., \( M(\hat{Q}_1, R_4) = \{\{f_y, f_m\}\}\). Furthermore, since constraint mapping is generally many-to-many, an emission can be a complex query (rather than a single constraint).

---

\(R_1) \{A1 \text{O X}; \text{SimpleMapping}(A1) \mapsto A2 = \text{AttrNameMapping}(A1); \text{emit}:[A2 \text{O X}]\)
\(R_2) \{[\text{ti}] = \text{L}; [f = F] \mapsto A = \text{LnFnToName}(L, F); \text{emit}:[\text{author} = A]\)
\(R_3) \{[\text{ti}] = \text{L} \mapsto \text{emit}:[\text{author} = L]\)
\(R_4) \{[\text{ti contains}] = P_1 \mapsto P_2 = \text{RewriteTextPat}(P_1); \text{emit}:[\text{ti-word contains}] = P_2\)
\(R_5) \{[\text{ti}] = \text{T} \mapsto \text{emit}:[\text{title starts}] = T\)
\(R_6) \{[\text{year}] = Y_1; [\text{month}] = M_1 \mapsto Y_2 = \text{NormYear}(Y_1); M_2 = \text{NormMonth}(M_1); D = \text{MonthYearToDate}(Y_2, M_2); \text{emit}:[\text{date during}] D\)
\(R_7) \{[\text{year}] = Y_1 \mapsto Y_2 = \text{NormYear}(Y_1); \text{emit}:[\text{date during}] Y_2\)
\(R_8) \{[\text{kwrd contains}] = K_1 \mapsto K_2 = \text{RewriteTextPat}(K_1); \text{emit}:[\text{ti-word contains}] = K_2 \lor [\text{subject-word contains}] = K_2\)
\(R_9) \{[\text{category}] = C \mapsto S = \text{MapCategoryTerms}(C); \text{emit}:[\text{subject}] = S\)

Figure 3: Mapping rules \( K_{Amazon} \) for Amazon.

---

\[^1\]We assume a target context based on the “power search” interface at www.amazon.com, with slight changes for the purpose of illustration.
For instance, rule $R_8$ 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 $R_6$ emits [pdate during Hay/97] (shown as $a_d$ in S1, Figure 2), we know that $S(f_y \land f_m) = a_d$, if $R_6$ is sound.

Furthermore, the matchings of a rule must be inseparable, i.e., a rule should handle only those truly-dependent constraints. In other words, the mapping rules effectively encode constraint dependencies. For instance, for $R_6$ the matching $\{f_y, f_m\}$ is indeed inseparable. Separating $f_y$ and $f_m$ would only result in a suboptimal mapping: Since Amazon requires that the year be specified in a given month, there is no mapping for only a month, i.e., $S(f_y) = \text{True}$. Thus, $S(f_y) \land S(f_m) = \text{True} = [\text{pdate during 97}]$, which is broader that the mapping $a_d$ obtained with $R_6$. For this reason, $R_6$ is a sound rule.

Based on the rule framework, Algorithm SCM (in Figure 4) translates simple conjunctions. The algorithm is relatively straightforward. First (in step 1), we evaluate the rules to find the matchings in a given query $\hat{Q}$. As discussed, this matching process effectively partitions $\hat{Q}$ into subsets of inseparable constraints. We then compute the emissions for those subsets as their mappings. The target query is simply the conjunction of all such emissions (step 3). In addition, we must remove submatchings to avoid redundancy (step 2). We can eliminate a matching if it is a subset of some other matching, because the latter will generate a “stricter” mapping with more “underlying constraints.” For instance, $R_6$ defines the mapping to pdate from both the original pyear and pmonth, while $R_7$ from only the former. Note that $R_7$ is useful to generate a partial date if pmonth is not constrained in the original query. However, for queries with both pyear and pmonth, such as $\hat{Q}_1$ in Figure 2, $R_7$ yields a redundant matching $\{f_y\}$, given the larger matching $\{f_y, f_m\}$ produced by $R_6$. We next illustrate Algorithm SCM with Example 4. Please refer to [17] for the proof that the algorithm does generate minimal subsuming mappings.

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

1. $A \leftarrow \bigcup [M(\hat{Q}_1, R_3), \ldots, M(\hat{Q}_1, R_8)] = \{\phi, \phi, \{f_1\}, \{f_1, f_3\}, \{f_1, f_9\}, \{f_1, f_3, f_9\}, \phi, \{f_y, f_m\}, \{f_y\}, \{f_k\}, \phi\} = \{m_3: \{f_1\}, m_4: \{f_1\}, m_6: \{f_y, f_m\}, m_7: \{f_y\}, m_8: \{f_k\}\}$

2. We remove the matching $m_7$ (of $R_7$), as $m_7 \subset m_6$ (of $R_6$). Thus, $A = \{m_3, m_4, m_6, m_8\}$.

---

**Algorithm SCM: Simple-Conjunction Mapping**

**Input:** $Q$: a simple-conjunction query in the original context; $K$: mapping rules for a target system $T$.

**Output:** $S(Q)$: minimal subsuming mapping w.r.t. $T$.

**Procedure:**

1. // find all the matchings for any rule in $K$:
   - $A \leftarrow M(\hat{Q}, K) \equiv \bigcup [M(\hat{Q}, R), \text{for all } R \in K]$.

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

3. output $S(Q)$ as the conjunction of all the remaining matchings in $A$.

---

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

3. The matchings $m_i$ are in $A$ map (by rule $R_3, R_4, R_6, R_8$) to target queries $a_n, a_{11}, a_d,$ and $a_{12} \lor a_{13}$ (shown on the right top of Figure 2). The output is their conjunction, i.e., $S(Q_1) = a_d \land a_{11} \land a_d \lor (a_{12} \lor a_{13}) = S_1$.

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

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

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

Input: $Q$: an arbitrary query in the original context; $K$: mapping rules for a target system $T$.
Output: $S(Q)$: minimal subsuming mapping w.r.t. $T$.

Procedure:
1. convert $Q$ into DNF $\tilde{Q} = \sum_{i=1}^{m} \tilde{D}_i$, where $\tilde{D}_i$ is a simple conjunction of constraints.
2. for each $\tilde{D}_i$: $S(\tilde{D}_i) \leftarrow SCM(\tilde{D}_i, K)$
3. return $S(Q) \equiv S(\tilde{Q}) = \sum_{i=1}^{m} S(\tilde{D}_i)$

Figure 5: Algorithm DNF.

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

5 DNF-based Scheme for Complex Queries

In this section we present a first translation algorithm for complex queries with arbitrary Boolean ($\wedge$, $\vee$) combination of constraints. Specific complications arise for such queries because of the implication of the Boolean operators. In particular, can the mapping $S(.)$ distribute over $\wedge$ and $\vee$?

In Example 2 we observed that conjuncts in $Q = \tilde{C}_1 \wedge \tilde{C}_2 = (\tilde{f}_1 \vee \tilde{f}_2) \wedge \tilde{f}_3$ 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 $\tilde{D}_1: (\tilde{f}_1 \wedge \tilde{f}_2) \vee \tilde{D}_2: (\tilde{f}_2 \wedge \tilde{f}_3)$, it turns out that disjuncts are always separable (according to the results of [15]). Thus, we can handle $\tilde{D}_1$ and $\tilde{D}_2$ independently, i.e., $S(Q) = S(\tilde{D}_1) \vee S(\tilde{D}_2)$.

Furthermore, as the disjuncts are simple conjunctions, their mappings can be handled with Algorithm SCM. Thus, $S(Q) = SCM(\tilde{D}_1, K_{Amazon}) \vee SCM(\tilde{D}_2, K_{Amazon})$. Since the calls to SCM fire rule $R_2$ to handle the mappings $\{\tilde{f}_1, \tilde{f}_3\}$ for $\tilde{D}_1$ and $\{\tilde{f}_2, \tilde{f}_3\}$ for $\tilde{D}_2$, $S(Q)$ becomes [author = "C1ancy, Tom"]$\vee$ [author = "K1ancy, Tom"]). Note that the result is indeed the minimal mapping possible.

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

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

6 Traversal-based Top-Down Query Mapping

This section discusses Algorithm TDQM, which performs constraint mapping in a top-down traversal of a query tree. Although not essential, for the purpose of explanation, we represent a query in a query tree, with interior $\wedge$ and $\vee$ nodes, and leaf constraints. Figure 6 shows a book query $\tilde{Q}_{book}$ that we will use as our running example. Recall that $\tilde{Q}$ means that the query is conjunctive, while $Q$ 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 $Q$ are disjunctive, i.e., $Q = \wedge(\tilde{C}_i)$. Similarly, disjuncts are conjunctive, i.e., $Q = \vee(\tilde{D}_i)$. Of course, at the leaves, both $\tilde{C}_i$ and $\tilde{D}_i$ 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 $S(.)$ over $\wedge$ and $\vee$, we eventually would only need to handle leaf constraints (with Algorithm SCM). Modulo the conjunction problem, this top-down process is essentially the intuition for Algorithm TDQM (Figure 7).

The major challenge in Algorithm TDQM is to effectively handle conjunctions, which we will explore in more detail in Section 7. In particular, for inseparable conjuncts, we want to partition them into some separable subsets. For instance, as we will see, $\tilde{Q}_{book}$ is not separable, i.e., $S(\tilde{Q}_{book}) \neq S(\tilde{C}_1) S(\tilde{C}_2) S(\tilde{C}_3)$. However, it turns out that only $\tilde{C}_2$ and $\tilde{C}_3$ are truly dependent; i.e., $S(\tilde{Q}_{book}) = S(\tilde{C}_1) S(\tilde{C}_2) S(\tilde{C}_3)$. With the partition of $\{\tilde{C}_1\}$ and $\{\tilde{C}_2, \tilde{C}_3\}$, the mapping can...
proceed directly to $C_1$, and we need to rewrite only the subtree $(C_2 \land C_3)$. 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 $\hat{Q}_{\text{book}}$ (Figure 6) for Amazon with the rules $K_{\text{Amazon}}$ (Figure 3). To begin with, since $\hat{Q}_{\text{book}}$ is conjunctive, we must figure out how to separate the conjuncts (or otherwise rewrite the whole query as with Algorithm DNF). Section 7.2 will discuss Algorithm PSafe specifically for conjunction partition. As we will see, $\text{PSafe}(\hat{Q}_{\text{book}}, K_{\text{Amazon}})$ returns two blocks $B_1 = \{C_1\}$ and $B_2 = \{C_2, C_3\}$, i.e., $S(\hat{Q}_{\text{book}}) = S(\land(B_1))S(\land(B_2))$.

We first handle block $B_1$. As it is a single-conjunct block, the mapping proceeds directly to $C_1$. Furthermore, since disjuncts are always separable, we can separate $f_1f_j, f_1k_1,$ and $f_2k_2$. Since they are all simple conjunctions (of one or more constraints), we can handle them with Algorithm SCM. In summary, by traversing the $C_1$ subtree, we obtain $S(\land(B_1)) = \text{SCM}(f_1f_j, K_{\text{Amazon}}) \lor \text{SCM}(f_1k_1, K_{\text{Amazon}}) \lor \text{SCM}(f_2k_2, K_{\text{Amazon}})$.

As for $B_2$, note that intuitively $(C_2 \land C_3)$ are not separable, because $C_2$ has a pyear constraint that can combine with either pmonth constraint in $C_3$ to fire rule $R_6$ in $K_{\text{Amazon}}$. For inseparable conjuncts, we must rewrite the subtree to continue the mapping. In particular, we can distribute the root $\lor$ over the next level $\lor$, and thus $B_2 = f_yf_{m_1} \lor f_yf_{m_2}$. Intuitively, by pushing down the problematic $\land$, we can eventually collect the dependent constraints in some simple conjunctions (e.g., $f_yf_{m_1}$ and $f_yf_{m_2}$). As we rewrite $\land(B_2)$ to a disjunctive form, the mapping can proceed to the new disjuncts, i.e., $S(\land(B_2)) = \text{SCM}(f_yf_{m_1}, K_{\text{Amazon}}) \lor \text{SCM}(f_yf_{m_2}, K_{\text{Amazon}})$. Thus, the complete mapping of $\hat{Q}_{\text{book}}$ is $S(\land(B_1))S(\land(B_2))$.

Observe that during tree traversal, our algorithm actually rewrites the query. In particular, $\hat{Q}_{\text{book}}$ is effectively converted to $(f_1f_j \lor f_1k_1 \lor f_2k_2) \lor (f_yf_{m_1} \lor f_yf_{m_2})$ so that dependent constraints are collected in simple conjunctions. Note that, in comparison, Algorithm DNF would require a global and blind conversion into DNF: $(f_1f_jf_yf_{m_1} \lor f_1f_jf_yf_{m_2} \lor f_1k_1f_yf_{m_1} \lor f_1k_1f_yf_{m_2} \lor f_2k_2f_yf_{m_1} \lor f_2k_2f_yf_{m_2})$. 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 $f_{m_1}$ in three of them).

As Example 6 informally illustrated, Algorithm TDQM traverses a given query tree to perform the mapping. We structure this tree traversal as a recursive procedure in Figure 7. The procedure differentiates three cases: At an $\lor$-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.

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

**Input:** $Q$: an arbitrary query in the original context; $K$: mapping rules for a target system $T$.

**Output:** $S(Q)$: minimal subsuming mapping w.r.t. $T$.

**Procedure:**

1. **Case-1: disjunctive $\lor$-node.**
   - if $Q = \lor([D_1, D_2, \ldots, D_n])$:
     - for each $D_i$: $S(D_i) \leftarrow \text{TDQM}(D_i, K)$ //recursive call.
     - return $S(Q) = \lor([S(D_1), \ldots, S(D_n)])$ //separate $D_i$.

2. **Case-2: conjunctive $\land$-node with some non-leaf children.**
   - else if $Q = \land([C_1, C_2, \ldots, C_n])$ with some non-leaf $C_i$:
     - $\hat{S} \leftarrow \text{PSafe}(Q, K)$ //partition $Q$ into separable blocks.
     - for each $B_i \in \hat{S}$:
       - $B \leftarrow \text{Disjunctivize}(\lor(B_i))$ //local query rewriting.
       - $S(\land(B)) \leftarrow \text{TDQM}(B, K)$ //recursive call.
       - return $S(Q) = \land_{B \in \hat{S}} S(\land(B))$ //Case-3: simple conjunctions; leaf or $\lor$ of some leaves.
   - else if $Q$ is a simple conjunction:
     - return $S(Q) = \text{SCM}(Q, K)$ //call Algorithm SCM.

**Function Disjunctivize($\lor(B)$):** rewrite to a disjunctive form.

- if $B = \lor([C_1, \ldots, C_n])$, and $C_i = \lor([D_{i_1}, \ldots, D_{i_m}])$ for $i = 1, \ldots, n$ then $\hat{B} \leftarrow \lor([C_1, \ldots, C_n])$ //single-conjunct block.
   - else: if $B$ distributes over $\lor$, e.g., $\lor(\lor(D_{i_1} \lor D_{i_2}), (D_{i_3} \lor D_{i_4}))$ //becomes $\lor(D_{i_1}D_{i_2}, D_{i_3}D_{i_4}, D_{i_5}D_{i_6})$
     - return $\hat{B} = \lor([D_{i_1}D_{i_2} \lor D_{i_3}D_{i_4} \lor D_{i_5}D_{i_6} : i_j \in [1 : m_i])$.

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

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

In particular, at a conjunction, we rewrite locally and incrementally each inseparable block into a disjunctive form. As Figure 7 (bottom) shows, function Disjunctivize converts a conjunctive subtree by distributing the $\land$ at the root over the $\lor$ at the next level. For instance, in Example 6 we rewrote $f_yf_{m_1} \lor f_yf_{m_2}$ to $(f_yf_{m_1} \lor f_yf_{m_2})$: the rewriting was localized to block $\{C_2, C_3\}$. 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 $(A \lor B)(C)$ into $(AC \lor BC)$, if the dependency is between $A$ and $C$, we need not to further rewrite $BC$ at all.

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

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

7.1 Safety Conditions for Conjunct Separability

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

Base Case: Simple-Conjunction Conjunctions

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

Example 7: Let $\hat{Q} = \hat{C}_1(f_j f_f) \land \hat{C}_2(f_y) \land \hat{C}_3(f_m)$ (part of the query in Figure 6). For rules $K_{Amazon}$ (Figure 3) representing target Amazon, $\hat{Q}$ is not separable because of the matching $\{f_y, f_m\}$ (for rule $R_3$), 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 $\{\text{date during May/97}\}$ (generated by $R_3$ from matching $\{f_y, f_m\}$). In fact, it will drop the month component, because with the separation $\mathcal{R}_7$ will fire instead (with matching $\{f_y\}$ from $\hat{C}_2$).

Definition 2 (Safety for Base-Case Conjunctions): Let $Q = C_1 \cdots C_n$, where $C_i$’s are simple conjunctions. $Q$ is safe w.r.t. rules $K$ if $\mathcal{M}(Q, K) - \cup_{i=1}^n \mathcal{M}(C_i, K) = \phi$; otherwise $Q$ is unsafe.

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

General Case: Disjunctive-Query Conjunctions

Conjunctions in our query trees generally have the form $Q = C_1 \cdots C_n$, where $C_i$’s are disjunctive with “ingredient” disjuncts $I_{ij}$, i.e., $C_i = I_{i1} \lor \cdots \lor I_{in_i}$. (The ingredients $I_{ij}$ can themselves be complex queries.) Since $C_i$’s are conjuncts, any combinations of their ingredients of the form $\hat{Q} = I_{i_1} \cdots I_{i_{nk}}$ is an implicit conjunction in $Q$. Intuitively, $Q$ is separable when there is no interdependencies among the ingredients from different $C_i$’s. In other words, when all such “ingredient conjunctions” are separable, i.e., $\mathcal{S}(\hat{Q}) = \mathcal{S}(I_{i_1}) \cdots \mathcal{S}(I_{i_{nk}})$, then $Q$ as the “whole conjunction” must also be separable, which we illustrate with Example 8.

Example 8: Suppose $Q = C_1 C_2 = (I_{11} \lor I_{12})(I_{21})$. (We can view $C_2$ as disjunctive with one disjunct.) To see the ingredient conjunctions, let’s convert $\hat{Q}$ into a disjunctive form with function Disjunctivize (Figure 7). That is, we compute $Q = \text{Disjunctivize}(\hat{Q}) = \lor\{\hat{D}_1: I_{11}, I_{21}, \hat{D}_2: I_{12}\}$. 

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., $\mathcal{S}(\hat{Q}) = \mathcal{S}(C_1)\mathcal{S}(C_2)$. Since $\hat{Q} = \hat{D}_1 \lor \hat{D}_2$, the left hand side $\mathcal{S}(\hat{Q})$ is equivalent to $\mathcal{S}(\hat{D}_1) \lor \mathcal{S}(\hat{D}_2)$ (disjuncts are always separable), or $\mathcal{S}(I_{11})\mathcal{S}(I_{21}) \lor \mathcal{S}(I_{12})\mathcal{S}(I_{21})$ because $\hat{D}_1$ and $\hat{D}_2$ are separable. The right hand side $\mathcal{S}(C_1)\mathcal{S}(C_2)$ is also equivalent to the last expression, since $\mathcal{S}(C_1) = \mathcal{S}(I_{11}) \lor \mathcal{S}(I_{12})$ and $\mathcal{S}(C_2) = \mathcal{S}(I_{21})$.

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

Definition 3 (Safety for General-Case Conjunctions):

Let $\hat{Q} = C_1 \cdots C_n$, where $C_i$’s are disjunctive, i.e., $C_i = I_{i1} \lor \cdots \lor I_{in_i}$, and $I_{ij}$’s are arbitrary queries. Let $Q = \text{Disjunctivize}(\hat{Q})$. $\hat{Q}$ is safe w.r.t. rules $K$ if all the disjuncts (as a conjunction $I_{i_1} \cdots I_{i_{nk}}$) in $Q$ 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 $\hat{Q} = C_1 C_2 = (x \lor y)(z)$. 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 \(S(x) = \text{True}\) and \(S(xz) = S(z)\)), we can show that \(S(Q) = S(C_1)S(C_2)\): First, \(S(Q) = S(xz \lor yz) = S(xz) \lor S(yz) = S(z) \lor S(yz)\). Thus we obtain \(S(Q) = S(z)\), since \(S(z) \geq S(yz)\). Now consider the mapping of the other way: \(S(C_1)S(C_2) = S(x \lor y)S(z) = [S(x) \lor S(y)]S(z)\). Thus \(S(C_1)S(C_2)\) also simplifies to \(S(z)\) since \(S(x) = \text{True}\). Therefore, \(Q\) is actually separable while being unsafe. Observe that this “anomaly” is solely because (the mapping of the unsafe term \(y(z)\) is “masked” by \(S(x) = S(z)\), which would not occur if \(S(x) \neq \text{True}\).

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

Testing the Safety Conditions

We next discuss how to efficiently test the safety conditions (to determine separability). In principle, to check if \(Q = C_1 \cdot \cdot \cdot C_n\) is safe, we can recursively apply Definition 3. As each application will “Disjunctivize” the query, eventually we will deal with the base case (when all the \(C_i\)’s become simple conjunctions) in Definition 2.

In fact, we can first convert \(C_i\)’s into DNF to avoid the recursion: Note that, in Definition 3, when all \(C_i\)’s are in DNF, the ingredients \(I_{ij}\) are just simple conjunctions. Therefore, we can check the safety of \(I_{i1_{k1}} \cdot \cdot \cdot I_{in_{kn}}\) with Definition 2. To illustrate, consider (in Figure 6) \(Q_m = C_1C_2C_3 = (f_{j1} \lor f_{j1} \lor f_{j2})(f_{j1} \lor f_{j2})\). Since \(C_i\)’s are already in DNF, we then check the safety for all the six conjunctions \(D_{i1}, D_{i2}, \cdot \cdot \cdot D_{i6}\). After applying Definition 2, we conclude that (among others) \(D_{i1}\) is unsafe with the cross-matching (for rules \(K_{Amazon}\) \{f_{j1}, f_{j2}\}). Thus, \(Q_m\) is unsafe.

However, this “brute-force” approach is not as efficient as possible; it unnecessarily relies on \(C_i\)’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 \(C_i\)’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 \(C_i\) the essential DNF (or EDNF), and write it as \(D_{i}(C_i)\). While omitting “useless” terms from \(C_i\) does not impact the safety results, in most cases it will greatly simplify the safety check. We illustrate by redoing the \(Q_m\) example.

Example 9 (Essential DNF): Consider again (in Figure 6) \(Q_m = C_1C_2C_3 = (f_{j1} \lor f_{j1} \lor f_{j2})(f_{j1} \lor f_{j2})\). The EDNF’s are \(D_{i}(C_1) = \epsilon, D_{i}(C_2) = f_{j2}, \text{and } D_{i}(C_3) = f_{j1} \lor f_{j2}\). Intuitively, only those “essential” constraints (i.e., \(f_{j2}, f_{j1}, \text{and } f_{j2}\)) involved in the potential cross-matchings (namely \(f_{j1}, f_{j1}\) and \(f_{j2}, f_{j2}\)) remain in the EDNF. Note that we use \(\epsilon\) to represent “something unimportant” (for testing safety) or “don’t care.”

Replacing each \(C_i\) by \(D_{i}(C_i)\), we can then check the safety with the simplified expression \((\epsilon)(f_{j2})(f_{j2})\). In turn, we will check the safety for simple conjunctions \(D_{i1} = (\epsilon)(f_{j2})(f_{j2})\) and \(D_{i2} = (\epsilon)(f_{j2})(f_{j2})\). Obviously, testing the safety for these \(D_{i1}\)’s involves less work than that for \(D_{i1}\)’s (based on the full DNF) just illustrated, because using \(C_i\)’s EDNF results in fewer and simpler terms. Note that we indeed obtain the same result that \(Q_m\) is unsafe, since all the cross-matchings (i.e., \(f_{j1}, f_{j1}\) and \(f_{j2}, f_{j2}\)) are preserved through the simplification.

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

7.2 Partitioning Conjunctive Queries

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

Specifically, when a conjunction \(C_1 \cdot \cdot \cdot C_n\) is safe, our algorithm simply returns the \(n\) blocks \(\{C_1\}, \ldots, \{C_n\}\), which means that every conjunct can be independently translated. Otherwise, for an unsafe conjunction, Algorithm \(PSafe\) can collect those inseparable conjuncts in the same block. This partition can limit the query structure conversion to within a block. Note that we can instead simply convert the whole unsafe conjunction into a disjunction (or even directly into DNF as in Algorithm \(DNF\)). However, such blind conversion is not necessary since not all the conjuncts in an unsafe conjunction are interrelated.

More formally, for a conjunction \(Q = C_1 \cdot \cdot \cdot C_n\), a partition \(P\) is a set of blocks \(B_1, \ldots, B_m\). Each block contains some conjuncts \(C_i\). For instance, for query \(Q_m\) (Example 9) the partition will have two blocks \(B_1 = \{C_1\}\) and \(B_2 = \{C_2, C_3\}\). We require that each conjunct \(C_i\) be handled in exactly one block (so that \(C_i\) does not repeat in the mapping). Note that the original conjunction can be written as \(Q = B_1 \cdot \cdot \cdot B_m\), where \(B_j\) is the conjunction of block \(B_j\) (i.e., \(B_j = \land(B_j)\)). For query mapping the partition must be safe, i.e., \(S(Q) = S(B_1) \cdot \cdot \cdot S(B_m)\). In our example, we can verify that \(S(Q) = S(B_1)S(B_2) = S(C_1)S(C_2)\). The problematic matchings \(\{f_{j1}, f_{j1}\} \text{ and } \{f_{j2}, f_{j2}\}\) are both contained in
In addition, we want the blocks to be *minimal*, i.e., no \( B_j \) can be further safely partitioned into smaller blocks. In our \( \tilde{Q}_{\text{book}} \) example, we cannot separate block \( \{ C_2, C_3 \} \).

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

**Example 10:** We continue Example 9 to partition conjunction \( \tilde{Q}_{\text{book}} \). In Section 9, we found two cross-matchings: \( m_1 = \{ f_y, f_{m1} \} \) and \( m_2 = \{ f_y, f_{m2} \} \). To partition \( \tilde{Q}_{\text{book}} \), we then find the blocks that cover the matchings: Since \( m_1 \) is a matching covered by \( C_2 \) and \( C_3 \), we consider \( B = \{ C_2, C_3 \} \) 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 \( C_1 \) does not participate in any cross-matchings, it is a block by itself. Therefore, the partition is \( \{ \{ C_1 \}, \{ C_2, C_3 \} \} \).

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

**Example 11:** Consider \( \tilde{Q}_a = C_1C_2C_3 = (x)(y)(yu \lor v) \). Assume that the matchings for constraints \( x, y, u, v \) are \( \{ x, y \}, \{ u \}, \) and \( \{ v \} \). Apparently, the partition needs blocks \( \{ C_1, C_2 \} \) and \( \{ C_1, C_3 \} \) as they both cover the matching \( \{ x, y \} \). This partition (that includes both blocks) is not minimal: It turns out that only \( \{ C_1, C_2 \} \) is necessary, i.e.,

\[
S(\tilde{Q}_a) = S(C_1C_2)S(C_3).
\]

In fact, we can verify that \( \tilde{Q}_a \equiv (x)(y)(u \lor v) \), and thus clearly we can separate \( C_1 \) and \( C_3 \).

To contrast, for the same constraints, consider another query \( \tilde{Q}_b = C_1C_2C_3 = (x)(y \lor u)(y \lor v) \). Again, the matching \( \{ x, y \} \) appears across \( C_1 \) and \( C_2 \) as well as \( C_1 \) and \( C_3 \). However, unlike the previous case, now both blocks \( \{ C_1, C_2 \} \) and \( \{ C_1, C_3 \} \) are required. Consequently, we will merge the overlapping blocks (so that \( C_1 \) will not be handled twice). Thus the partition is the single block \( \{ C_1, C_2, C_3 \} \), i.e., \( S(\tilde{Q}_b) = S(C_1C_2C_3) \).

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

### 8 Optimality, Compactness, and Complexity

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

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

We also note that, while carefully addressing constraint dependencies, our algorithm is quite efficient. In fact,
when a query does not involve dependent constraints, our algorithm pays virtually no extra cost (in addition to the mapping of single constraints). Recall that we address dependencies among conjuncts by checking the safety conditions. As Section 7.1 discussed, we can check the safety for \( \hat{Q} = C_1 \cdots C_n \) brute force by first converting each \( C_i \) 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 \( C_i \) and \( k \) the (maximal) number of constraints in each \( C_i \). 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 9). In other words, \( C_i \)’s EDNF will only contain those constraints that participate in potential matchings spanning beyond \( C_i \). If \( e \) is the number of those “essential” constraints remaining, the EDNF of \( C_i \) will have an upper bound of \( 2^e \) terms. Multiplying all such terms from different \( C_i \)’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 \( C_i \)’s are simply \( \epsilon \) (e.g., \( D_\epsilon(C_i) = \epsilon \) in Example 9). Therefore, we only need to check one term (i.e., \( 2^e = 1 \)) consisting of all \( e \); 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 TDOM pays virtually no extra cost when no constraint dependencies exist.

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

References


