Chapter 4

An Automated Query-Translation Methodology

An expressive and abstract query model for clinical medicine is only a prerequisite to providing applications with access to heterogeneous clinical databases. The second component of the technology is a principled methodology for translating automatically queries specified using the abstract query model to semantically equivalent queries that can be executed by heterogeneous clinical databases. In this chapter, I describe the query-translation methodology of the TransFER model. In Section 4.1, I discuss the desirable features of query-translation methodologies in general and I introduce the framework of the TransFER query-translation methodology, which is based on the concept of attribute grammars and the relational algebra. In Section 4.2, I describe how database programmers can use a mapping language based on the relational algebra to encode site-specific mappings between an FER schema and a relational database. Section 4.3 provide a description and a detailed example of how the translation methodology applies the encoded mappings within the attribute-grammar framework to translate ReFER queries to site-specific SQL queries. In Section 4.4, I describe the theoretical basis of the optimization techniques used by the TransFER compiler to simplify translated queries.

4.1 Framework of the Methodology

An ideal query-translation methodology for the FER query model would satisfy the following criteria:

1. *Automation*: The methodology translates queries with no human intervention. Although human intervention is required to specify the

mappings between a particular clinical database and the abstract FER query model, subsequent use of the mappings to translate individual queries is completely automated. The human labor required to specify the mappings is amortized over the queries that are later translated automatically.

2. *Latitude*: The methodology is general and powerful enough to translate FER queries to the query models of a great variety of clinical databases, provided that the databases are semantically consistent with the FER query model. *Semantic consistency* means, informally, that the same semantic information is represented in the query model of the implemented database as is represented in the abstract query model, although the technical means by which the information is represented may differ. Differences between semantically consistent query models constitute *representational heterogeneity* [Barsalou and Gangopadhyay 1992]. Several types of representational heterogeneity exist [Agarwal 1991] and the ideal query-translation methodology is able to resolve all of them.

3. *Efficiency*: The methodology automatically generates queries that are as computationally efficient as queries translated manually by database programmers. Different formulations of semantically equivalent queries can dramatically influence the queries' computational efficiency, even given the query-optimization capabilities of many commercial relational query engines. Database programmers, who are familiar with the semantics of particular databases and with the costs of various query operations, are adept at writing queries that retrieve requested data with the minimum computation. The ideal automated translation methodology applies similar optimization principles to generate queries of comparable efficiency.

The query-translation methodology that I propose, called TransFER, attempts to fulfill each of these design criteria. The methodology provides automated query translation by implementing a syntax-directed translation engine that takes as input only a ReFER query and a set of database-specific mappings. The methodology provides an expressive database mapping language to accommodate a wide variety of clinical database implementations. The

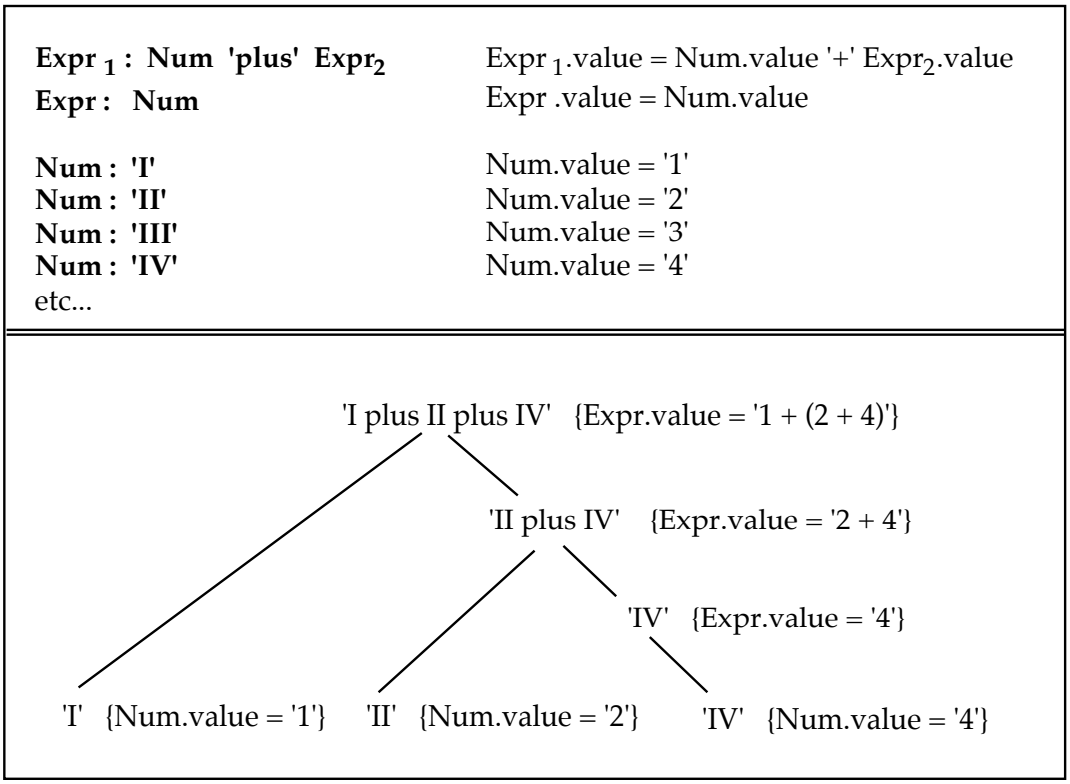
methodology optimizes the queries that it automatically generates by applying well-founded transformation techniques based on the theory of relational algebra.

The computational framework of the translation methodology consists of an *attribute grammar* for the ReFER query language (the language was described in Section 3.3). Attribute grammars, first introduced by Knuth [Knuth 1968], are context-free grammars (CFGs) in which each terminal and nonterminal symbol is associated with a set of attributes. Attribute grammars are useful for formally representing the transformation of one language to another because the set of attributes associated with the syntactic symbols of one representation system can denote the semantics of the symbols in a different representation system. When an expression is parsed by an attribute grammar, the values of the attributes associated with each symbol are assigned, usually based on the values of attributes associated with adjacent symbols in the parse tree. The rules for assigning values to attributes are called *semantic functions*. Together, the productions of a CFG and the semantic functions for deriving the attributes of each symbol in the CFG constitute an attribute grammar.

4.1.1 A sample attribute grammar

As a general example, Figure 4.1 shows a simple attribute grammar that recognizes expressions denoting the addition of Roman numerals using the operator "plus," such as "II plus IV." The purpose of the attribute grammar is to translate these expressions into conventional mathematical notation, such as "2 + 4," so that they can be computed by a standard calculator The boldfaced text in the upper panel of Figure 4.1 represents the CFG for the Roman numeral addition language. The plain text in the upper panel of Figure 4.1 represents the semantic functions for assigning a value to each attribute. Note that each symbol on the left-hand side of a grammar rule (that is, "Expr" and "Num") has a single attribute named "value," which denotes the value of the symbol in conventional mathematical notation. For terminal symbols (that is, leaves in the parse tree), such as "Num," the semantic function assigns a constant value to the attribute; for nonterminal symbols, such as "Expr," the semantic function derives a value for the attribute based on the values of attributes previously assigned to symbols on the right-hand side of the grammar rule. The lower panel of Figure 4.1 shows

the parse tree for the expression "I plus II plus IV," with all attributes assigned. Note that the attribute value at the root of the parse tree ("1 + 2 + 4") has precisely the same semantics as the input string ("I plus II plus IV"), although it is represented in a completely different syntactic notation. The effect of the attribute grammar, therefore, is to translate any expression formulated in the Roman numeral addition language into a semantically equivalent expression in conventional mathematical notation. If one were to interface the attribute grammar in Figure 4.1 to a standard calculator, one could enter the expression "I plus II plus IV" and receive the result "7."



semantically equivalent expressions formulated in the relational algebra. By translating ReFER queries to relational algebra, one can retrieve the requested data from a relational database, just as by translating Roman numeral expressions to mathematical notation one can compute the result with a standard calculator. By customizing the attribute grammar to particular relational database implementations, one can generate relational algebra translations of ReFER queries that are site specific. The relational algebra that I use is a version of the standard relational algebra, which I have enhanced to have the expressive power of SQL[1]. I describe the algebra in Section 4.1.3.

Figure 4.2 shows an excerpt of attribute grammar for the ReFER query language and a parse tree generated by the attribute grammar for the simple ReFER query

"[name-of(pt) | (pt : Patient)]".

The translation of the query is performed by the semantic functions of the ReFER attribute grammar, just as the translation in Figure 4.1 is performed by the semantic functions of the attribute grammar for the Roman numeral addition language. The semantic functions corresponding to each symbol in the ReFER attribute grammar (shown in plain text in the upper panel of Figure 4.2) assign a relational algebra expression to the symbol at the time it is parsed, much the way that the semantic functions of the attribute grammar in Figure 4.1 assign a conventional arithmetic expression to each symbol at the time it is parsed. For terminal symbols, such as Entity and Relationship, the relational algebra expression corresponding to the symbol on the left-hand side of a grammar rule is constant; these expressions, in fact, represent the site-specific mappings for the entity types and relationship types of the FER schema. For example, the attribute grammar in Figure 4.2 assigns the relational algebra expression *P* to the entity type Patient and the relational algebra expression *N* to the relationship type Person-Name. Note that *P* and *N* are just place holders in the example of Figure 4.2; in an actual ReFER attribute grammar, they might be complex relational algebra expressions, such as

---

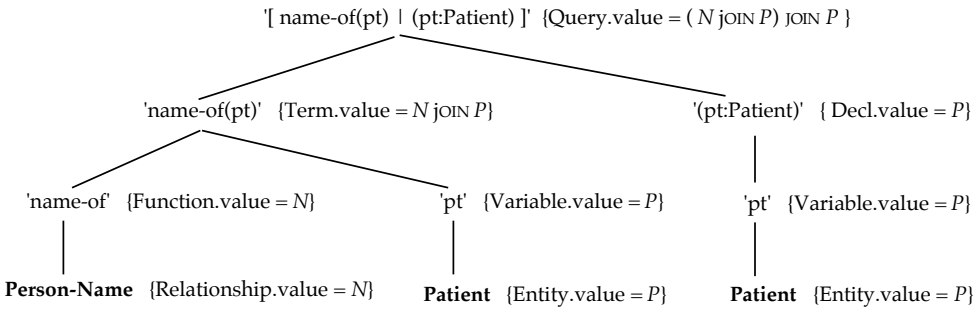
[1]I refer here to the ANSI standard SQL published in1989[Date 1989a], commonly referred to as "SQL89."

```
Project( [ LastName, FirstName ],
        Select( [ Ward = "Medicine" OR
                  Ward = "Surgery"],
                Union (  InPatients, OutPatients ) ) ).
```

The specification of mapping expressions like *P* and *N* is discussed in Section 4.2. Note that variables in ReFER queries are assigned the values of the entity types with which they are associated in the declaration statement (that is, "pt" is assigned the value for "Patient") and functions in ReFER queries are assigned the values of the relationship types to which they correspond in the FER schema (that is, "name-of" is assigned the value for "Person-Name." The rationale for these assignments is discussed in Section 4.3.3.2. For nonterminal symbols, such as "Term" and "Query," the corresponding relational algebra expressions must be constructed from the relational algebra expressions previously assigned to the symbols on the right-hand side of the grammar rules. The semantic functions of the attribute grammar construct relational algebra expressions by specifying relational algebra operations, such as JOIN and UNION. Note that the semantic functions for the nonterminal symbols "Query" and "Term" in Figure 4.2 are oversimplified because they do not specify the fields on which relational algebra expressions are to be joined. This omission is to simplify the diagram only. The semantic functions of the actual attribute grammar do specify the join fields, as described in Section 4.3.3.

Because the relational algebra is an *applicative* language [MacLennan 1987], the application of relational operators to relational expressions produces new relational expressions. By recursively combining relational algebra expressions as specified by the semantic functions, the attribute grammar constructs a single relational algebra expression in a bottom-up fashion. This relational algebra expression is semantically equivalent to the input ReFER query, just as the arithmetic expression constructed at the root of the parse tree in Figure 4.1 is equivalent to the input Roman numeral expression. The relational algebra expression subsequently can be translated to SQL via a straightforward transformation, as I discuss in Section 4.3.4. When the attribute grammar is interfaced to an SQL query engine, ReFER queries can retrieve data directly from relational databases.

Query : '[' Factor '|' Decl ']'	Query .value = Term.value JOIN Decl.value
$Term_1$: Function '(' $Term_2$ ')'	$Term_1$ .value = Function.value JOIN $Term_2$.value
Term : Variable	Term.value = Variable.value
Decl : '(' Variable ':' Entity ')'	Decl.value = Variable.value
Variable : <string>	Variable.value = Entity.value
Function : <string>	Function.value = Relationship.value
Entity : Patient	Entity.value = *P*
Relationship : Person-Name	Relationship.value = *N*



ReFER attribute grammar, all table names and field names referenced in the relational algebra expression that is produced by the translation will exist in the local relational database, assuming that the mappings have been specified correctly. Examples of mappings for entity types and relationship types appear in Section 4.2.

2. TransFER attribute grammars do not immediately evaluate the relational algebra expressions that they generate. Deferral of evaluation allows the resulting expressions to be transformed into more efficient forms (while retaining the same semantics) before the relational query is actually executed. For example, an optimizer may determine that joining the relational algebra expression *N* to the expression *P* twice in the translation result of Figure 4.2 is superfluous, and that the simplified expression (*N* join *P*) is equivalent to the original expression ((*N* join *P*) join *P*). Relational algebra is well suited for representing the translation of ReFER queries because numerous formal techniques already exist for optimizing relational algebra expressions. The use of these techniques is discussed in Section 4.4.

To summarize, the TransFER attribute grammar translates a ReFER query into a semantically equivalent relational query by (1) decomposing the ReFER query into its syntactic components, (2) translating each component into an appropriate relational algebra expression, and (3) combining the expressions into a single relational algebra query. Before I discuss how the semantic functions of the attribute grammar generate and combine relational algebra expressions to effect query translations, I introduce the relational algebra language in which those expressions are formulated. Recall that this language is used to represent the site-specific mappings for FER entity and relationship types (such as "Patient" and "Person-Name") as well as the site-independent translations of ReFER query operations (such as "name-of(pt)").

4.1.3 The extended relational algebra (ERA)

Because the goal of the TransFER attribute grammar is to translate ReFER queries into representations that are meaningful with respect to relational databases, it is natural to use a relational data-manipulation language (DML) to specify the relational semantics of ReFER query constructs. The most prominent DMLs for

relational databases are the relational calculus and the relational algebra [Codd 1972]. Although variants of the relational calculus (usually SQL) constitute the query interfaces of most modern relational databases, I have chosen the relational algebra for representing the relational semantics of ReFER constructs. The relational algebra is preferable because it is an *applicative* language, which supports both the incremental construction of relational queries from modular subcomponents and the systematic optimization of the constructed queries. The incremental assembly of relational subcomponents is compatible with the framework for query translation that I described earlier in this section because it allows the relational semantics of FER entity types and relationship types to be specified in a context-free fashion and to be combined later as needed. The systematic optimization of relational algebra expressions is consistent with the efficiency criterion for query translation methodologies because it allows optimization procedures to be encoded and automatically performed. The relational algebra and the relational calculus are equally expressive [Codd 1972, Klug 1982], and conversion from one to the other is straightforward.

The relational algebra is a query language based on the notion of applying operators to relations. The original relational algebra and extensions to it have been described extensively [Codd 1970, Klug 1982, Maier 1983]. I have adopted this work to define an *extended relational algebra* (ERA) that I use to represent the relational semantics of FER schemas and ReFER queries. As opposed to the original relational algebra, ERA is equal in expressiveness to SQL[2] and can, therefore, utilize the full capabilities of modern SQL query processors.

Table 4.1 lists the operators of the ERA. The syntax for relational operators that I use in this document differs from the conventional syntax but the semantics of the operators are similar. Let <relation> be any well-defined relational algebra expression with respect to the relational data model, <attrs> be any set of field names corresponding to the columns in a relation, <consts> be a set of constant values, and <conds> be a set of conjoined conditions of the form "<attrs> = <attrs>" or "<attrs> = <consts>." Given these definitions, the relational algebra operations of the ERA are listed in Table 4.1.

---

[2]with the sole exception that the ERA does not include the SQL sorting operation "ORDER BY."

ERA Operator	Syntax
PROJECT	**Proj ([**<attrs>**],** <relation>**)**
MULTIPROJECT	**MProj ([**<attrs>**],** <relation>**)**
SELECT	**Sel ([**<conds>**],** <relation>**)**
JOIN	**Join ([**<conds>**],** <relation>, <relation> **)**
UNION	**Union (** <relation>, <relation>**)**
DIFFERENCE	**Diff (** <relation>, <relation>**)**
AGGREGATE	**Agg([** <arg-attrs>, <group-attrs>, <fxn>, <new-attrs>**],** <relation>**)**
RENAME	**Rename([**<attr-renames>**],** <relation> **)**
EXTEND	**Extend([** <attr-defns>**],** <relation> **)**
EXISTS	**Exists(** < relation> **)**

Table 4.1: The operators of the extended relational algebra. The definitions of <relation>, <attrs>, <conds>, <fxn>, <attr-renames>, and <attr-defns> are discussed in the text. Note that the syntax differs from the conventional syntax for relational operators (that is, π, σ, $\cup$, etc.), but is more convenient for expository purposes.

4.1.3.1 Project, Select, Join, Union, Difference

The operators PROJECT, SELECT, JOIN, UNION, and DIFFERENCE in Table 4.1 have the same semantics as specified by Codd for the original relational algebra [Codd 1972]. The correspondence between the syntax of the operators used in Table 4.1 and the conventional syntax is as follows:

ERA Syntax	Conventional Syntax
Proj ([<attrs>]**,** <relation>**)**	$\pi_{<\text{attrs}>}$ <relation>
Sel ([<conds>]**,** <relation>**)**	$\sigma_{<\text{conds}>}$ <relation>
Join ([<conds>]**,** <relation>, <relation>**)**	<relation> $\bowtie_{<\text{conds}>}$ <relation>
Union (<relation>**,** <relation>**)**	<relation> $\cup$ <relation>
Diff (<relation>**,** <relation>**)**	<relation> $-$ <relation>

4.1.3.2 Multiprojection

The MULTIPROJECT operator is identical to the PROJECT operator except that duplicate rows are retained (the classic PROJECT operator eliminates any duplicate rows that result from "projecting out" certain fields in a table).

4.1.3.3 Aggregation

The AGGREGATE operator is a generalization of the relational aggregation operator introduced by Klug [Klug 1982]. The AGGREGATE operator in Table 4.1 partitions a relation into sets of tuples that have the same values of the attributes specified in <group-attrs>; within each partition, it applies the aggregation function <fxn> to the set of values specified in <arg-attrs>, and returns a new relation consisting of the Cartesian product of the result of the aggregation function (which receives the new attribute name <new-attrs>) and the value of <group-attrs> for that partition. For example, given the relation "Doctors" shown in Figure 4.3 and the aggregation function AVERAGE, the relational algebra expression

R = **Agg(** [{Age}, {Specialty}, AVERAGE, {Avg-Age}]**,** Doctors**)**

would generate the relation R shown in Figure 4.4 by taking the average of the "Age" values for each partition with the same value of "Specialty" and naming the resulting value "Avg-age."

Doctors =

Name	Specialty	Age
"Smith"	"Ortho"	50
"Jones"	"Ortho"	60
"Thomas"	"Cardio"	40
"Cross"	"Cardio"	50

Figure 4.3: A relational table named "Doctors."

R =

Specialty	Avg-Age
"Ortho"	55
"Cardio"	45

Figure 4.4: A relational table generated by applying the AGGREGATE operator to the relational table in Figure 4.3.

4.1.3.4 Rename and Extend

The RENAME operator simply renames the specified attributes that appear in the relation to which it is applied. The renaming specifications appear in a list of terms of the form "<old-attr-name> => <new-attr-name>." For example, given the relation shown in Figure 4.3, the relational algebra expression

S = **Rename([** Specialty **=>** Dept, Name **=>** Last-Name **],** Doctors**)**

would generate the relation S shown in Figure 4.5.

S =

Last-Name	Dept	Age
"Smith"	"Ortho"	50
"Jones"	"Ortho"	60
"Thomas"	"Cardio"	40
"Cross"	"Cardio"	50

Figure 4.5: A relational table generated by applying the RENAME operator to the relational table in Figure 4.3.

The EXTEND operator adds a column to the relation to which it is applied. The value of the column may be a constant or the value of any other column that already exists in the relation. The extension specifications appear in a list of terms of the form "<new-attr-name> == <new-attr-value>." For example, given the relation shown in Figure 4.3, the relational algebra expression

T = **Extend([** Degree == "MD"**,** Experience == Age**],** Doctors**)**

would generate the relation T shown in Figure 4.6.

T =

Name	Specialty	Age	Degree	Experience
"Smith"	"Ortho"	50	"MD"	50
"Jones"	"Ortho"	60	"MD"	60
"Thomas"	"Cardio"	40	"MD"	40
"Cross"	"Cardio"	50	"MD"	50

Figure 4.6: A relational table generated by applying the EXTEND operator to the relational table in Figure 4.3.

4.1.3.5 Exists

The EXISTS operator actually does not perform any transformation on the relation to which it is applied, but specifies that any JOIN operations subsequently applied to the Exists() expression are *semi-joins* [Ullman 1988 p. 60]. For example, given the relations Doctors and Fathers shown in Figures 4.3 and 4.7, respectively, the relational algebra expression

U = **Join([** Name = LName**],** Doctors**, Exists(**Fathers**))**

would generate the relation U shown in Figure 4.8.

Fathers =

LName	Child-Name
"Smith"	"Maggie"
"Smith"	"Chuck"
"Cross"	"Carlos"
"Cross"	"Martha"

Figure 4.7: A relational table called "Fathers."

U =

Name	Specialty	Age
"Smith"	"Ortho"	50
"Cross"	"Cardio"	50

Figure 4.8: A relational table generated by applying the JOIN operator to the relational table in Figure 4.3 and to an EXISTS operation over the relational table in Figure 4.7.

4.1.3.6 Virtual relations

An unusual but useful feature of the extended relational algebra is the notion of *virtual relations*. Virtual relations are named relations that have attributes and may appear in relational algebra expressions, but which do not actually exist as stored relations in a database. A virtual relation represents a pair of *functions* that simulate the existence of a stored relation at the time that the virtual relation is joined to an actual relation. For example, consider the virtual relation "Age-BirthYear," which has the following schema:

```
Age-BirthYear = field        functional definition
                curr-age:    compute_age(birth-yr)
                birth-yr:    compute_birth_year(curr-age).
```

The virtual relation represents mappings between birth years and current chronological ages. For example, on November 1, 1995, the age 50 maps to the birth year 1945 and the birth year 1960 maps to the age 35 (note that these mappings are not strictly precise because the exact birth date is not considered; the functions make the assumption that every person's birth date is January 1). Because Age-BirthYear is a virtual relation, the mappings between ages and birth years are not actually stored. Rather, the functions "`compute_age()`" and "`compute_birth_year()`" that correspond to the fields of the virtual relation compute the appropriate mappings when the virtual relation is joined to an actual relation. For example, the mappings are invoked in the following relational algebra expression, which derives the birth years of the physicians represented in Figure 4.3.

Proj([birth-yr **],**
Join([Age = curr-age **],**
Doctors,
Age-BirthYear **)).**

The mechanism by which the mapping is simulated is that the value of the field to which "curr-age" is joined (that is, "Age") is substituted as the argument to the function "`compute_birth_year()`**"** and the result is assigned to the field "birth-yr." In this case, the result of joining the "Doctors" relation to the virtual

relation "Age-BirthYear" is equivalent to extending the "Doctors" relation with the computed birth year:

```
Proj( [ birth-yr ],
    Extend( [ birth-yr == compute_birth_year(Age) ],
        Doctors ).
```

This is the case only because there is a one-to-one association between ages and birth years. In general, however, the functions defined for virtual relations may be multivalued, so that the mapping specified by the virtual relation cannot be implemented by extending each row with a single value. Therefore, the join is required.

The advantage of representing functional mappings with virtual relations is that a query optimizer may manipulate virtual relations and actual relations similarly. Specifically, joins between virtual relations and actual relations may be re-ordered or deferred by an optimizer so that the costs of executing the functions specified in the virtual relations are reduced. For example, if a select operation greatly reduces the size of an actual relation by excluding many rows, a join between the actual relation and a virtual relation should be deferred until after the select operation to reduce the number of times the function is executed.

Formally, every virtual relation, R_v, defines two disjoint sets of fields, A and B, such that the schema of R_v = AB. A mutual functional dependency exists between A and B such that A = f(B) and B = g(A), that is, each value of A in R_v functionally determines the corresponding values of B and each value of B in R_v functionally determines the corresponding values of A. Note that, in general, A and B may each constitute more than one field. The functions f and g are executable functions that map from values of A to sets of values of B and vice versa. The important constraint exists that virtual relations can be joined to actual relations only on all the values of A or all the values of B. This constraint ensures that all of the arguments to the functions that implement the virtual relation are bound when the functions are executed (recall that the values of the fields on which the virtual relation is joined are substituted into the arguments of the functions).

In practice, many useful virtual relations represent functions that have no inverses. An example is the virtual relation "DOB-Age," which maps a precise date-of-birth to a current age in years:

```
DOB-Age =       field        functional definition
                curr-age:    compute_age(dob)
                dob:         <no function>.
```

Because any age in years implies a *range* of possible birth dates, there is no function that can map from an age to a single DOB. However, the useful operation of mapping from a DOB to a single age is possible and is performed by the function "compute_age()." The implication of this asymmetry is that the virtual relation "DOB-Age" may be joined to actual relations only on the field "dob." The virtual relation may not be joined on the field "curr-age." A query optimizer that reorders joins involving virtual relations must be aware of this constraint. For example, given the tables defined in Figures 4.3 and 4.9, the following are valid and invalid relational algebra expressions:

valid:
```
Proj( [ curr-age ],                          (see Figure 4.9)
    Join( [ DOB = dob ],
        Patient,
        DOB-Age) )
```

invalid:
```
Proj( [ dob ],                               (see Figure 4.3)
    Join( [ Age = curr-age ],
        Doctors,
        DOB-Age) ).
```

4.1.3.7 Examples

The complete syntax of the Extended Relational Algebra appears in Appendix A. Examples of relational algebra expressions encoded in the syntax of Table 4.1 appear in the following list (the semantics of these expressions are based on the relational schema shown in Figure 4.9):

Patient

Name	Phone#	DOB
John	(415) 723-7633	12-2-57
Mary	(510) 325-8790	3-25-62

Doctor

MD-Name	Phone#	DOB
Colin	(415) 494-8767	1-7-39
Gordo	(510) 320-6966	11-17-58
Pam	(510) 390-8754	4-23-49

Patient-Doc

Patient	Doctor
John	Gordo
John	Pam
Mary	Colin
Mary	Pam

5. The names and ages of all patients:

```
Proj( [ Name, curr-age ],
      Join( [ DOB = dob ],
            Patient,
            DOB-Age) )
```

   (This expression assumes the definition of the virtual relation "DOB-Age" as specified above.)

6. The names and phone numbers of patients who are associated with at least one physician:

```
Proj( [Name, Phone#],
      Join( [ Name = Patient ],
            Patient,
            Exists( Patient-Doc ) ) )
```

7. The number of physicians associated with each patient:

```
Agg( [Doctor, Name, COUNT, Doc-Count],
      Join( [ Name = Patient ],
            Patient,
            Patient-Doc ) )
```

Expressions in the ERA language are generated by the semantic functions of the TransFER attribute grammar. The ERA expressions represent the relational translations of ReFER language constructs in queries parsed by the attribute grammar. The *site-specific* semantic functions generate the ERA translations of FER entity types and relationship types. I discuss these semantic functions in the next section. The *site-independent* semantic functions combine these ERA translations according to the syntax of a particular ReFER query to generate a single ERA expression that constitutes the relational translation of the query. I discuss these semantic functions in Section 4.3.

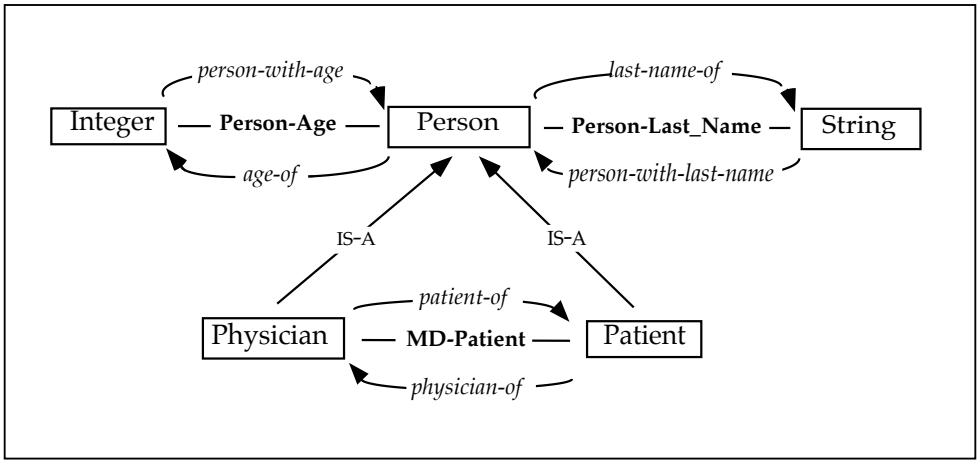
4.2 Mapping FER Schemas to Relational Databases

As I discussed in Section 4.1.2, certain of the semantic functions in the TransFER attribute grammar are site specific. These semantic functions define translations

from ReFER expressions to ERA expressions that vary depending on the clinical database implementation at a particular site. Due to local modeling decisions, relational databases that store similar clinical information may vary from site to site in numerous respects, including the naming of tables and fields, the degree of normalization, the representation of type hierarchies, and the composition of controlled vocabularies. Therefore, the relational algebra translations of certain ReFER expressions also must vary from site to site to represent the semantics of the ReFER expressions accurately. For example, consider the ReFER query Q1, which is specified with respect to the FER schema in Figure 4.10:

Q1: [last-name-of(p) | (p : Person) AND age-of(p) > 55]

Given the semantics of the FER schema and the ReFER query language, the query Q1 requests the last names of all persons in the database over the age of 55.



2. The entity types of the type hierarchy Person-Patient-Physician are represented in one table at Site A but are distributed across two tables at Site B.

3. Persons are uniquely identified by the combination of their first and last names at Site A, whereas persons are uniquely identified by their social security numbers at Site B.

4. Each person's age is explicitly stored at Site A, whereas each person's age must be computed from the stored date of birth at Site B.

5. The associations between patients and their physicians are represented in a separate table solely for this purpose at Site A, whereas the associations are represented in the same table as patients are represented at Site B (which must be denormalized because the patient–physician association is many-to-many).

Despite these differences, note that the information content specified by the FER schema in Figure 4.10 is completely represented in both of the relational schemas. The relational schemas at sites A and B are, therefore, *semantically consistent* with the FER schema (according to the informal definition of semantic consistency that I proposed in Section 4.1).

Site A

PERSON

fname	lname	age	type
Phil	Smythe	38	pt
Carol	Flynn	33	pt
Colin	Krup	56	md
Gordo	Matic	37	md
Pam	Simmons	46	md

PHYSICIAN_ROSTER

pt-fname	pt-lname	md-fname	md-lname
Phil	Smythe	Gordo	Matic
Phil	Smythe	Pam	Simmons
Carol	Flynn	Colin	Krup
Carol	Flynn	Pam	Simmons

Site B

PATIENT

pt-ssn	first	last	dob	md-ssn
149584223	Phil	Smythe	12-2-57	249633805
149584223	Phil	Smythe	12-2-57	136420428
216426077	Carol	Flynn	3-25-62	186757471
216426077	Carol	Flynn	3-25-62	136420428

PHYSICIAN

md-ssn	first	last	dob
186757471	Colin	Krup	1-7-39
249633805	Gordo	Matic	11-17-58
136420428	Pam	Simmons	4-23-49

Note that the set of entity types and relationship types that appears in a TransFER attribute grammar depends on the FER schema for which the attribute grammar is defined. For example, the attribute grammar for the FER schema in Figure 4.10 would include the entity type symbols "Person," "Patient," and "Physician" and the relationship type symbols "Person-Last_Name," "Person-Age," and "MD-Physician" (the built-in entity types Integer and String are included by default).

The semantic functions for the entity types and relationship types of an FER schema assign values to the *attributes* of the entity-type and relationship-type symbols in the TransFER attribute grammar. In other words, each entity-type and relationship-type symbol has a set of attributes, the values of which are assigned by the semantic functions. Specifically, entity-type symbols have the attributes "ERA" and "Id," and relationship-type symbols have the attributes "ERA," "Id," and "Fid." The values of these attributes instruct the compiler how to assemble a relational algebra expression at the local database that is equivalent to the corresponding entity type or relationship type in the FER schema. In this section, I introduce the formal method by which the attributes specify these instructions; this method constitutes the *mapping language* of TransFER. In Appendix A, I present the complete specifications of the mapping language and I characterize the power of this language to map FER schemas to a variety of heterogeneous relational database schemas. When a ReFER query is parsed, the attribute grammar assembles the relational algebra expressions that correspond to the entity and relationship types within the query according to the site-specific mappings and combines these expressions to generate a complete relational algebra translation of the query (as described in Section 4.1.2).

4.2.1 Entity-type mappings

Recall that in the FER data model, entity types represent sets of unique objects in the domain of discourse. In order to map this concept to a relational database, the semantic function for each entity-type symbol, E, assigns an extended relational algebra expression, R, to the attribute "ERA" such that *each row in R represents a unique instance* of the entity type E. For example, the following are the assignments of ERA for the entity types of the FER schema in Figure 4.10 and the relational database implementations at sites A and B:

Site A

Person ::= ERA = " PERSON"
Patient ::= ERA = "Sel([type = "pt"], PERSON)"
Physician ::= ERA = "Sel([type = "md"], PERSON)"

Site B

```
Person ::= ERA = "Union( Proj([ssn],
                         Rename([pt-ssn -> ssn],PATIENT)),
                    Proj( [ssn],
                         Rename([md-ssn -> ssn],PHYSICIAN) ) )"
```

Patient ::= ERA = "Proj([pt-ssn], PATIENT)"
Physician ::= ERA = "PHYSICIAN".

Additionally, the semantic function for each entity-type symbol, E, assigns a list of field names, F_I, to the attribute "Id" such that F_I denotes the *key* of the relational algebra expression for the entity type E. By the definition of "keyness," each element of F_I must be a field in the corresponding ERA expression, R, and the concatenated values of F_I for each row of R must be unique. For example, the following are the assignments of the attribute "Id" for the entity types of the FER schema in Figure 4.10 and the relational database implementations at sites A and B:

Site A

Person ::= Id = [lname,fname]
Patient ::= Id = [lname,fname]
Physician ::= Id = [lname,fname]

Site B

Person ::= Id = [ssn]
Patient ::= Id = [pt-ssn]
Physician ::= Id = [md-ssn].

Several points regarding the mappings for Person, Patient, and Physician at sites A and B are noteworthy:

1. The SELECT operator is required in the ERA values of the entity types Patient and Physician at site A because the instances of Patient and Physician are represented together in the table `PERSON`.

2. The UNION operator is required in the ERA value of the entity type Person at site B because the instances of Person are distributed across the tables `PATIENT` and `PHYSICIAN`.

3. The RENAME operator is required in the ERA value of the entity type Person at site B because the field names denoting the keys of the tables `PATIENT` and `PHYSICIAN` are different in each table (that is, "`pt-ssn`" and "`md-ssn`"); because a single field name is required as the value of Id for Person, the keys must be "unified" before the tables are unioned.

4. The PROJECT operator is required in the ERA value of the entity type Patient at site B because the table `PATIENT` is denormalized and therefore may contain multiple rows for each Patient entity; the PROJECT operator eliminates duplicates and ensures that the Id value, `[pt-ssn]`, is unique.

Together, the values of the attributes "ERA" and "Id" values for each entity-type symbol completely specify how that entity type is represented in a particular relational database.

4.2.2 Relationship-type mappings

In the FER data model, relationship types represent sets of unique objects that associate pairs of entities. In order to map this concept to a relational database, the semantic function for each relationship-type symbol, A, assigns an extended relational algebra expression, R, to the attribute "ERA" such that each row in R represents a unique instance of the association denoted by the relationship type A. Because each instance of a relationship type in the FER data model represents an association between two entities, the semantic function also assigns a paired list of field names, $<F_1, F_2>$, to the attribute "Fid;" F_1 and F_2 denote the fields of the relational algebra expression R that contain the identifiers of the entities associated by each row of R. Each field list, F_1 and F_2, is annotated with the appropriate entity type of the entity that it represents. For example, the

following are the assignments of ERA and Fid for the relationship types of the FER schema in Figure 4.10 and the relational database implementations at sites A and B (it is useful to review the FER schema in Figure 4.10 and the relational database schemas in Figure 4.11 when inspecting these specifications):

Site A

```
Person-Last_Name ::=  ERA = " PERSON"
                      Fid = { Person([lname, fname]),
                              String([lname]) }
Person-Age ::=    ERA = " PERSON"
                  Fid = { Person([lname, fname]),
                          Integer([age]) }
MD-Patient ::= ERA = "PHYSICIAN-ROSTER"
                   Fid = { Physician([md-lname, md-fname]),
                           Patient([pt-lname, pt-fname]) }
```

Site B

```
Person-Last_Name::=
          ERA = "Union( Proj([ssn, last],
                            Rename([pt-ssn -> ssn], PATIENT)),
                        Proj( [ssn, last],
                            Rename([md-ssn -> ssn], PHYSICIAN) ) )"
           Fid = { Person([ssn]),
                   String([lname]) }
Person-Age::=
       ERA = "Join([dob = date-of-birth],
                   Union( Proj([ssn, dob],
                            Rename([pt-ssn -> ssn], PATIENT)),
                          Proj( [ssn, dob],
                            Rename([md-ssn -> ssn], PHYSICIAN) ) ),
                   DOB-AGE)"
        Fid = { Person([ssn]),
                Integer([age]) }
MD-Patient ::=
       ERA = "PATIENT"
       Fid = { Physician([md-ssn]),
               Patient([pt-ssn]) }.
```

Note how the value of Fid for the relationship type MD-Patient at site A consists of one field list annotated by "Physician" and one field list annotated by "Patient" because the relationship type MD-Patient associates instances of the entity type Physician with instances of the entity type Patient . In each row of the relational algebra expression for MD-Patient, the respective values of the fields [md-lname, md-fname] and [pt-lname, pt-fname] represent the identities of the physician and the patient associated by that instance of the relationship type. Several other points regarding the mappings for Person-Last_Name, Person-Age, and MD-Patient at sites A and B are noteworthy:

1. At site A, one of the lists of field names in the Fid value is a subset of the other list. This is appropriate because the field [lname] that represents each person's last name in the table PERSON is part of the key of that table, [lname, fname], which represents each person's identity. In other words, the last name associated with each person at site A effectively can be derived from the unique identifier of the person.

2. At site B, the UNION operator is required in the ERA values of the relationship types Person-Last_Name and Person-Age because those relationship types associate *all* instances of Person (whether they are patients or physicians) with their names and their ages.

3. Because site B does not explicitly store the ages of patients and physicians, the ERA value of the relationship type Person-Age requires a join to the virtual table DOB-AGE in order to derive the age (Section 4.1.3.6 discusses virtual tables). DOB-AGE is a virtual table that implements the function "compute_age()," which computes the difference in years between an arbitrary date and the current date. The virtual table "DOB-AGE" and the function "compute_age()" must be defined at the time the mappings are specified. For example, the following definition of DOB-AGE is consistent with the mapping for Person-Age at site B:

```
DOB-Age = field              functional definition
          date-of-birth:     <no function>
          age:               compute_age(date-of-birth).
```

Because relationships are themselves objects in the FER data model, each mapping for a relationship type must denote the unique identifier of the relationship instances. Therefore, the semantic function for each relationship-type symbol, A, assigns a list of field names, F_I, to the attribute "Id" such that F_I denotes the *key* of the relational algebra expression for the relationship type A. The same constraints apply to the Id values of relationship types as apply to the Id values of entity types. The Id values of relationship types are important when the relationship types themselves participate in relationship types, as the FER data model allows. The following are the assignments of Id for the relationship types of the FER schema in Figure 4.10 and the relational database implementations at sites A and B:

Site A

Person-Last_Name ::= Id = [`lname,fname`]

Person-Age ::= Id = [`lname,fname,age`]

MD-Patient ::= Id = [`md-lname,md-fname,`
`pt-lname,pt-fname`]

Site B

Person-Last_Name ::= Id = [`ssn,last`]

Person-Age ::= Id = [`ssn,age`]

MD-Patient ::= Id = [`pt-ssn,md-ssn`].

Together, the ERA, Fid, and Id values for each relationship-type symbol completely specify how that relationship type is represented in a particular relational database.

The mappings for entity-type and relationship-type symbols, which are defined at each site in the manner I have just described, assign values to the terminal symbols of a TransFER attribute grammar, thereby "customizing" the attribute grammar to a particular site. The assigned values, which consist of relational algebra expressions (the ERA values) and information about those expressions (the Id and Fid values), subsequently may be combined and manipulated by the site-independent semantic functions of the attribute grammar to produce relational translations of arbitrary ReFER queries. In the following section, I introduce the site-independent semantic functions and describe how they translate valid ReFER expressions to semantically equivalent SQL queries.

4.3 Translating ReFER queries to SQL queries

Recall from Section 3.3.7 that any ReFER query of the form

Query ::= '[' Termlist '|' Decllist 'AND' Predicate ']',

in which Termlist, Decllist, and Predicate conform to the specifications of the ReFER language, has the formal meaning

$$\bigcup_{b \in B(\text{Decllist, Predicate}),} \nu[\text{Termlist, b}]$$

where B(Decllist, Predicate) = { b : b ∈ B(Decllist) and τ[Predicate, b] = TRUE}.

Informally, the definition states that the result of a ReFER query is the multiunion of the values of the term list computed for each binding that is specified in the declaration list and that fulfills the logical condition of the predicate. I will refer to the term list, the declaration list, and the predicate that make up a ReFER query as the Termlist, Decllist, and Predicate *components* of the query, respectively. For example, the ReFER query

"[last-name-of(pt), last-name-of(md) | (pt : Patient), (md : Physician) AND
MD-Patient(md,pt) AND
age-of(pt) > 55"

returns the last names of each patient–physician pair in the database such that the physician is treating the patient and the patient is older than 55. The Termlist component is "last-name-of(pt), last-name-of(md)," the Decllist component is "(pt : Patient), (md : Physician)," and the Predicate component is "MD-Patient(md,pt) AND age-of(pt) > 55."

To translate the ReFER query accurately to an SQL query, TransFER must reproduce the semantics embodied in the formal definition of the ReFER query in a corresponding relational algebra expression. TransFER subsequently must translate the relational algebra expression to SQL. I discuss the first step in Sections 4.3.1–3 and the second step in Section 4.3.4.

4.3.1 Translating to the relational algebra: an overview

A ReFER query is translated to a relational algebra expression by the site-independent semantic functions of the TransFER attribute grammar, which combine the site-specific mappings for entity types and relationship types to reproduce the meaning of any ReFER query. The site-independent semantic functions perform this translation by first specifying relational algebra *subexpressions* for the Termlist, the Decllist, and the Predicate components of the ReFER query; the semantic functions subsequently combine the subexpressions by applying the JOIN operator. The result is a relational algebra expression that reproduces the formal meaning of the ReFER query. I will describe how the semantic functions specify the appropriate subexpressions for the Termlist, Decllist, and Predicate components of a ReFER query in Sections 4.3.2 and 4.3.3. In this section, I describe informally the relational algebra subexpressions generated for the Termlist, Decllist, and Predicate components, and I describe informally how the subexpressions are combined to reproduce the semantics of the original ReFER query. I hope to provide the reader with a high-level understanding of the translation process before I delve into the details of the translation mechanism.

4.3.1.1 Translations of ReFER query components

The meanings of the relational algebra subexpressions that are generated by the semantic functions for the Termlist, the Decllist, and the Predicate components of a ReFER query may be described informally as follows:

1. *Decllist*: For the Decllist component, the semantic functions generate the *Cartesian product* of the relational algebra mappings for each entity type that appears in the declaration list. For example, the Decllist component

   "(pt : Patient), (md : Physician)"

   of the sample query contains the entity types Patient and Physician; therefore, the semantic functions generate the Cartesian product of the relational algebra mappings for the entity type Patient and the entity type Physician. Recall that the mapping for each entity type specifies a relational algebra expression (that is, a relational table) such that each

row of the table represents one instance of the mapped entity type (see Section 4.2.1). I shall call these tables "mapping tables." Each row of the mapping table for an entity type, therefore, represents one *binding* of the variable that is declared to range over that entity type. The Cartesian product of the mapping tables for each entity type in the declaration list, therefore, constitutes the Cartesian product of the bindings defined by each declaration in the declaration list. This construction reproduces precisely the semantics of Decllist in the ReFER query language (see Section 3.3.3).

2. *Predicate*: For the Predicate component, the semantic functions generate a relational algebra expression that contains one row for each binding of the variables in the predicate for which the predicate holds. For example, the Predicate component

   "MD-Patient(md,pt) AND age-of(pt) > 55"

   of the sample query results in a relational algebra expression that contains one row for each binding of the variables "md" and "pt" such that the physician bound to "md" is treating the patient bound to "pt" and the age of the patient bound to "pt" is greater than 55. Note that by *not* containing any rows that represent bindings of "md" and "pt" for which the predicate fails, the relational algebra expression effectively "filters out" all such bindings. This expression, therefore, reproduces the semantics of Predicate in the ReFER query language, as defined in Section 3.3.6. I describe the way in which the semantic functions of the attribute grammar filter out inappropriate bindings in Section 4.3.3.2.

3. *Termlist*: For the Termlist component, the semantic functions generate a relational algebra expression that contains the values of the term list computed for *all* bindings of the variables that appear in the term list. In other words, the semantic functions perform no filtering of variable bindings, although such filtering may be specified in the Predicate clause of the query. For example, the Termlist component

   "last-name-of(pt), last-name-of(md)"

of the sample query results in a relational algebra expression that contains one row for each pair of bindings of "pt" and "md"; the row contains the unique identifiers of the patient and the physician entities to which the variables are bound and the last names associated with these patient and physician entities. At a database site where social security numbers are used to identify persons, the Termlist of the sample query might result in the following relational table:

pt-ssn	pt-last-name	md-ssn	md-last-name
1234	Smythe	5678	Collins
5432	Brooks	9876	Matic
1234	Smythe	4444	Collins

Note that the table corresponding to the term list contains one row for every binding of "pt" and "md" although the query in which the term list appears includes the predicate "age-of(pt) > 55," which specifies that only certain bindings of "pt" should be used to compute the results of the query. The filtering specified by the predicate is applied to the term list later.

4.3.1.2 Combining the translations of ReFER query components

Given the meanings of the relational algebra subexpressions generated for the Termlist, Decllist, and Predicate components of a ReFER query, the semantic functions of the attribute grammar reproduce the meaning of the entire ReFER query by combining the subexpressions in the following manner:

1. The subexpression for the Decllist is joined to the subexpression for the Predicate. Because the Decllist subexpression contains one row for each binding of the declared variables whereas the Predicate subexpression contains rows for only those bindings that satisfy the predicate, the effect of the join is to produce the *intersection* of the set of all bindings and the set of bindings that satisfy the predicate. The resulting expression contains one row for each binding that (1) is defined by the declaration list and (2) satisfies the predicate; this set of bindings is, in fact, equivalent to the set of bindings for which the Termlist should be

evaluated according to the semantics of the ReFER query language (as defined in Section 3.3.7):

$$\{ b : b \in B(\text{Decllist}) \text{ and } \tau[\text{Predicate}, b] = \text{TRUE}\}.$$

For example, in the sample query, the relational subexpression for the Decllist "(pt : Patient), (md : Physician)" is joined to the relational subexpression for the Predicate "MD-Patient(pt,md) AND age-of(pt) > 55." The subexpression for the Decllist contains the Cartesian product of all Patient and Physician entities, whereas the subexpression for the Predicate contains only the pairs of Physician and Patient entities such that the entities are associated by the MD-Patient relationship type and the Patient entity is over the age of 55. The subexpressions are joined on the respective fields in each expression that contain the identifiers of the Physician and Patient entities (the semantic functions of the attribute grammar keep track of these sets of field names for each expression, as described in Section 4.3.2.1). Recall that the join of two tables simply drops all rows in either table that do not have corresponding rows in the other table. In this manner, the join "filters out" all of the bindings defined by the declaration list that do not satisfy the predicate. A join of the subexpressions for the Decllist and the Predicate of the sample query, therefore, produces the pairs of Patient and Physician entities such that the Physician entity is treating the Patient entity and the Patient entity has an age greater than 55. I call the relational algebra expression produced by joining the Decllist and the Predicate subexpressions the "Range." The Range expression contains precisely the set of bindings for which the term list should be evaluated according to the semantics of the ReFER query language.

2. The expression for the Termlist is joined to the Range expression. Because the Termlist expression contains the values of the term list computed for all bindings, whereas the Range expression contains only those bindings that satisfy the predicate, joining the expressions produces the values of the term list computed for only those bindings that satisfy the predicate. For example, in the sample query, the relational subexpression for the Termlist clause

"last-name-of(pt), last-name-of(md)"

is joined to the Range subexpression, which contains only those pairs of bindings for "pt" and "md" that satisfy the Predicate. The subexpressions are joined on the respective fields in each expression that contain the identifiers of the Patient and the Physician entities (the semantic functions of the attribute grammar keep track of these field names for each expression). Again, the effect of the relational join is to "filter out" all of the rows in the Termlist expression containing bindings of "pt" and "md" that do not appear in the Range expression. The resulting expression contains the values of the term list computed only for the set of variable bindings that appear in the Range expression. This is almost the correct result of the query, but one more operation is required.

3. The values of the Termlist expression are multiprojected. Recall that, for the sample query, each row of the Termlist expression contains the bindings of "pt" and "md" as well as the pair of last names associated with those bindings. The meaning of the query, however, is represented by only the pair of last names, so the multiproject operator must be applied to return only those fields. A multiprojection is specified rather than a projection because the formal meaning of ReFER queries is specified as the *multiunion* of generated values (see Section 3.3.7).

   For example, the result of the sample query is computed by multiprojecting only the fields "pt-last-name" and "md-last-name":

pt-last-name	md-last-name
Smythe	Collins
Brooks	Matic
Smythe	Collins

   The multiprojection is the final step in the translation process and the resulting relational algebra expression accurately represents the semantics of the ReFER query.

By first generating relational algebra subexpressions with the appropriate meanings for the Termlist, Decllist, and Predicate components of a ReFER query and by subsequently joining these subexpressions on the appropriate fields, the semantic functions of the attribute grammar succeed in reproducing the semantics of the original ReFER query. In the following section, I describe in detail how the semantic functions of the attribute grammar generate the correct relational algebra subexpressions for the Termlist, Decllist, and Predicate components and how the semantic functions manage the names of the relevant fields in each subexpression.

4.3.2 Site-independent semantic functions

The site-independent semantic functions of a TransFER attribute grammar combine the relational algebra expressions defined by the site-specific semantic functions to generate relational translations of arbitrary ReFER queries. Like the site-specific semantic functions, each site-independent semantic function generates a relational algebra expression (that is, table) and relevant information about that expression. Unlike the site-specific semantic functions, the site-independent semantic functions are not customized for specific relational database schemas; the same set of semantic functions appears in the TransFER attribute grammar at every site. The site-independent semantic functions are also different from their site-specific counterparts in that they may specify expressions that include *references to the attributes of grammar symbols assigned by other semantic functions*. For example, the following semantic function for the grammar symbol "Predicate" includes references to the attributes of the grammar symbols "Term" and "Op" (the context-free grammar rule for the symbol "Predicate" appears first, followed by the semantic function for that symbol's attribute "Predicate.ERA."):

Grammar rule: Predicate ::= $Term_1$ Op $Term_2$

Semantic function: Predicate.ERA = Join([$Term_1$.Id Op.value $Term_2$.Id],
$Term_1$.ERA,
$Term_2$.ERA)

Note that the arguments to the relational algebra JOIN operator are not relational tables, fields, and comparators (as the syntax of the relational algebra specifies),

but *references* to the attributes of grammar symbols on the right-hand side of the grammar rule. The values of these attributes represent relational tables, fields, and comparators; they must be computed recursively by the semantic functions for the grammar symbols "Term" and "Op." I will show how these references are used to generate valid relational algebra expressions in Section 4.3.3.

4.3.2.1 The attribute "Vid"

In Section 4.2, I described how site-specific semantic functions assign values to the attributes "ERA," "Id," and "Fid" for grammar symbols that denote entity types and relationship types. Site-independent semantic functions assign values to the same attributes for all of the other symbols in the ReFER attribute grammar (such as "Term," "Predicate," and "Decl"). Additionally, site-independent semantic functions assign values to an attribute named "Vid." The value of the Vid ("variable Id") attribute denotes the fields of a relational algebra expression that represent the *bindings* of the variables that appear in the corresponding ReFER expression. The value of Vid is a *set of pairs*, in which each element has the form "(var, field-list)," where "var" denotes a variable and "field-list" denotes the relational fields that represent the bindings of that variable. For example, in Section 4.3.1.1, I presented the following table to represent the relational translation of the Termlist clause "last-name-of(pt), last-name-of(md)" for a database site in which social security numbers are used to identify persons:

pt-ssn	pt-last-name	md-ssn	md-last-name
1234	Smythe	5678	Collins
5432	Brooks	9876	Matic
1234	Smythe	4444	Simmons.

The table contains the last names of pairs of Patients and Physician entities. The value of the Vid attribute for this relational algebra expression is

{ (pt, [`pt-ssn`]), (md, [`md-ssn`]) },

because the ReFER term list "last-name-of(pt), last-name-of(md)" contains the variables "pt" and "md," and the bindings of these variables in the relational table are represented by the values of the fields "`pt-ssn`" and "`md-ssn`," respectively. By denoting which fields of a relational algebra expression represent the bindings of the variables that appear in the corresponding ReFER

expression, the TransFER attribute grammar can filter out unwanted bindings by specifying SELECT and JOIN operations that involve those fields. This capability is critical for reproducing the semantics of ReFER queries in relational algebra expressions.

4.3.2.2 Site-independent semantic functions: an example

I now introduce the structure and operation of site-independent semantic functions through an example. The following semantic functions generate the relational translation of ReFER predicates that compare two terms, such as "pt = md" (again, the context-free grammar rule for the symbol appears first, followed by the semantic functions that define each attribute of the symbol):

Grammar rule: Predicate ::= $Term_1$ Op $Term_2$

Semantic functions:
Predicate.ERA = Join([$Term_1$.Id Op.value $Term_2$.Id],
$Term_1$.ERA,
$Term_2$.ERA)
Predicate.Id = ∅
Predicate.Vid = $Term_1$.Vid ∪ $Term_2$.Vid
Predicate.Fid = ∅

The grammar rule parses the ReFER predicate "pt = md" because "pt" and "md" are variables (which are terms) and "=" is a valid comparison operator. The semantic function for the attribute "ERA" specifies a JOIN operation that references the Id and ERA attributes of the two terms that are compared in the predicate. The semantic functions that define the Id and ERA attributes of these two terms may themselves reference the attributes of other terms. The recursion terminates when the grammar reaches a symbol that represents an entity type or a relationship type, because the semantic functions for entity-type and relationship-type symbols specify operations over the *base* tables and fields of a relational database (see Section 4.2). The semantic function for the attribute "Vid" computes the union of the Vid values for the two terms that are compared by the predicate, thereby producing a set of field lists that represents the bindings of all variables that appear in the ReFER predicate. For example, if the Vid value of $Term_1$ were { (pt, [`pt-ssn`]) } and the Vid value of $Term_2$ were { (md, [`md-ssn`]) }, the union of these values would produce the Vid value { (pt,

[pt−ssn]), (md, [md−ssn]) }. Lastly, the semantic functions for the attributes "Id" and "Fid" indicate that the values of these attributes are NULL because predicates (unlike terms) have no unique identifiers or foreign identifiers.

As an example of how the semantic functions operate, consider the relational translation of the ReFER expression "pt = md".

The grammar rule parses this expression as a Predicate clause, with "pt" as $Term_1$, "md" as $Term_2$, and "=" as Op. The semantic functions for the nonterminal symbols "Term" and "Op" assign appropriate values to each attribute of "pt," "md," and "=" (the semantic functions for "Term" and "Op" appear in Appendix B). Let us assume that the semantic functions have assigned the following attribute values for the expressions "pt," "md," and "=" (these assignments are consistent with the entity-type mappings for the relational schema Site B in Figure 4.11, assuming that the variable "pt" is declared to range over the entity type Patient and the variable "md" is declared to range over the entity type Physician):

"pt": Term.ERA = Proj([pt−ssn], PATIENT)
Term.Id = [pt−ssn]
Term.Vid = { (pt, [pt−ssn]) }
Term.Fid = ∅

"md": Term.ERA = PHYSICIAN
Term.Id = [md−ssn]
Term.Vid = { (md, [md−ssn]) }
Term.Fid = ∅

"=": Op.value = "="

When the attribute grammar substitutes the referenced values of the attributes into the semantic function definitions, the semantic functions generate the following relational algebra translation for the predicate "pt = md":

"pt = md": Predicate.ERA = Join([pt−ssn = md−ssn],
Proj([pt−ssn], PATIENT),
PHYSICIAN)
Predicate.Id = ∅

Predicate.Vid = { (pt, [`pt-ssn`]), (md, [`md-ssn`]) }
Term.Fid = ∅

Informally, the relational algebra translation of the predicate "pt = md" joins the table representing the Patient entity type with the table representing the Physician entity type on those fields that uniquely identify Patient and Physician entities in the respective tables; the result is a table that contains only rows in which the field that represents the binding of the variable "pt" equals the field that represents the binding of the variable "md"; that is, the table contains only rows representing pairs of Patient and Physician entities such that the Patient entity is the same as the Physician entity. In other words, the relational algebra translation of "pt = md" generates all the bindings of "pt" and "md" such that the predicate "pt = md" is true. This set of bindings is used later in the query-translation process to filter out the bindings for which the predicate does not hold (as described in Section 4.3.1). Note that the relational database at Site B in Figure 4.11 contains no persons who are both physicians and patients, so that no bindings "pt" and "md" satisfy the predicate "md = pt"; the relational algebra expression generated by the semantic functions for the predicate "pt = md," therefore, will return no rows at Site B.

4.3.2.3 Expanding predicates in relational algebra expressions

We have seen that site-independent semantic functions for the attribute "ERA" may define conditions in JOIN expressions that reference the Id attributes of other grammar symbols. For example, the JOIN condition in the semantic function

Predicate.ERA = Join([$Term_1$.Id Op.value $Term_2$.Id],
$Term_1$.ERA,
$Term_2$.ERA)

references the Id attributes of $Term_1$ and $Term_2$. In general, JOIN conditions in site-independent semantic functions may compare the Id, Vid, or Fid attributes of grammar symbols. Because the values of these attributes, in general, may be *lists* of field names (rather than a single field name, as was the case in Section 4.3.2.2), the values of the attributes (such as [`fname,lname`]) cannot simply replace *syntactically* the corresponding references to these values (such as $Term_1$.Id) because the results may not be valid relational algebra expressions. For example,

if $Term_1$.Id = [`fname,lname`] (a list of two fields) and $Term_2$.Id = [`first,last`] (also a list of two fields), the syntactic substitution of $Term_1$.Id and $Term_2$.Id with their corresponding field lists in the condition [$Term_1$.Id = $Term_2$.Id] produces the condition

[`fname,lname` = `first,last`].

This expression is not a valid JOIN condition in the syntax of relational algebra. To generate valid relational algebra JOIN conditions, the semantic functions must *expand* comparisons of attribute values that consist of multiple field names into *conjunctions* of comparisons for each pair of corresponding field names:

[`fname,lname` = `first,last`] $\Longrightarrow$ [`fname` = `first` and `lname` = `last`].

The semantic functions perform a similar expansion for cases where the values of Fid attributes are compared. Because the values of Vid attributes may be *sets* of field lists, in which each field list is associated with a variable, the expansion of Vid values is a little different. Specifically, the semantics of the predicate

[$Term_1$.Vid = $Term_2$.Vid]

specify that the field lists associated with each variable that *is common to both* Vid values are compared, and that the predicate expressions for each such comparison are conjoined into a single predicate. For example, if

$Term_1$.Vid ={ (pt, [`fname,lname`]), (md, [`first,last`]) } and
$Term_2$.Vid ={ (pt, [`Fst_name,Lst_name`]) },

then the expression [$Term_1$.Vid = $Term_2$.Vid] is expanded into

[`fname` = `Fst_name` and `lname` = `Lst_name`],

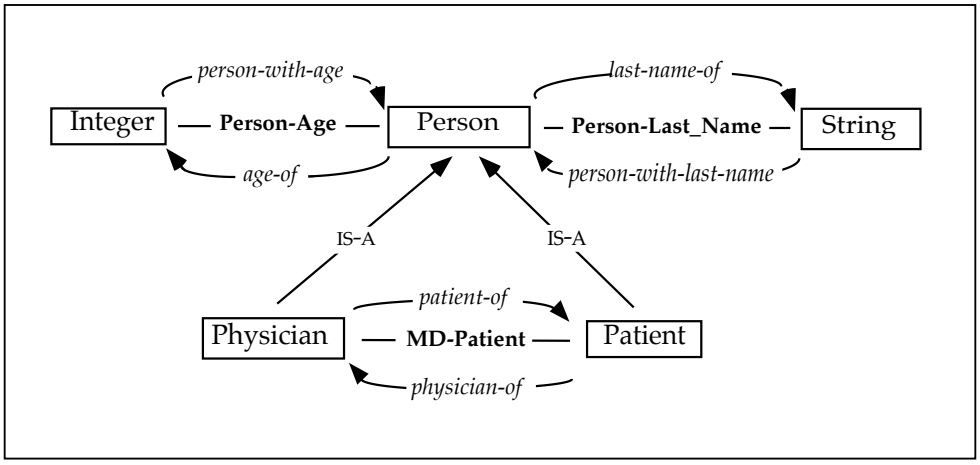
because the Vid values for $Term_1$ and $Term_2$ have the variable "pt" in common, and the field lists associated with the variable "pt" are [`fname,lname`] and [`Fst_name,Lst_name`], respectively. The rationale for these expansion rules will become evident with the sample query translation that appears in the next section.

4.3.3 A sample translation

I will show how the site-independent semantic functions of the TransFER attribute grammar perform query translations in general by tracing the translation of a sample ReFER query at two different database sites. The sample ReFER query requests the last names of all patients who are over the age of 35:

[last-name-of(pt) | (pt : Patient) AND age-of(pt) > 35].

The sample query is formulated with respect to the FER schema in Figure 4.12 (recall that the entity type Patient inherits the relationship types Person-Age and Person-Last_Name).



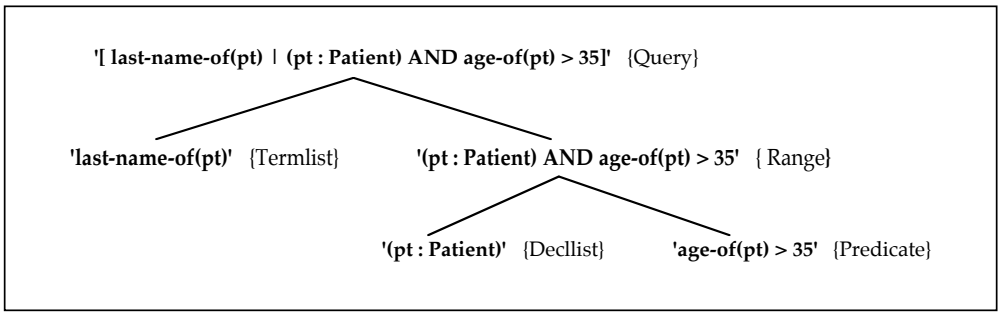


Figure 4.13: A portion of the parse tree for a sample ReFER query The context-free grammar of the ReFER query language decomposes the query into a Termlist, Decllist, and Predicate component (labeled in braces). The combination of the Decllist and Predicate components constitutes the Range component.

In the following sections, I will show how the semantic functions of the attribute grammar generate the appropriate relational algebra expressions for each of the three components and how the semantic functions combine the expressions to reproduce the semantics of the ReFER query. I will trace the operation of the TransFER attribute grammar with respect to each of the relational databases that appear in Figure 4.14. Note that the relational schemas in Figure 4.14 are the same schemas for which I formulated site-specific mappings in Section 4.2. The site-independent semantic functions that I describe in the following sections will use these site-specific mappings to generate the correct relational translations at each site for the Termlist, Decllist, and Predicate components of the query.

Site A

PERSON

fname	lname	age	type
Phil	Smythe	38	pt
Carol	Flynn	33	pt
Colin	Krup	56	md
Gordo	Matic	37	md
Pam	Simmons	46	md

PHYSICIAN_ROSTER

pt-fname	pt-lname	md-fname	md-lname
Phil	Smythe	Gordo	Matic
Phil	Smythe	Pam	Simmons
Carol	Flynn	Colin	Krup
Carol	Flynn	Pam	Simmons

Site B

PATIENT

pt-ssn	first	last	dob	md-ssn
149584223	Phil	Smythe	12-2-57	249633805
149584223	Phil	Smythe	12-2-57	136420428
216426077	Carol	Flynn	3-25-62	186757471
216426077	Carol	Flynn	3-25-62	136420428

PHYSICIAN

md-ssn	first	last	dob
186757471	Colin	Krup	1-7-39
249633805	Gordo	Matic	11-17-58
136420428	Pam	Simmons	4-23-49

The relational algebra expression (that is, ERA value) generated by the semantic functions for each declaration is equivalent to the relational algebra expression specified in the mapping for the entity type that appears in the declaration. For example, the relational algebra translation of the declaration "(pt : Patient)" is equivalent to the relational mapping of the entity type Patient. The semantic functions generate a Vid value for each declaration that associates the declared variable with the key fields of the relational algebra translation for the entity type in the declaration; this association specifies that the binding of the declared variable in each row of the relational algebra expression is expressed by the values of the key fields. As a side effect, the semantic functions also enter the association between the variable and its declared entity type in a symbol table so that other semantic functions that manipulate the variable can determine the entity type to which it corresponds.

When a declaration list consists of only one declaration, there is no need to compute a Cartesian product; the relational algebra expression for the declaration list is identical to the expression for the declaration. This equivalence is represented by the following semantic functions:

Grammar rule: Decllist ::= Decl

Semantic functions:
Decllist.ERA = Decl.ERA
Decllist.Id = Decl.Id
Decllist.Vid = Decl.Vid
Decllist.Fid = Decl.Fid

Given these definitions and the mappings for the entity type Patient specified in Section 4.2, the semantic functions generate the following relational algebra translations for the declaration list "(pt : Patient)" at Site A and Site B:

Site A

"(pt : Patient)" :
Decllist.ERA = Sel([`type` = "`pt`"], `PERSON`)
Decllist.Id = ∅
Decllist.Vid = { (pt, [`lname`, `fname`]) }
Decllist.Fid = ∅

Site B

"(pt : Patient)" : Decllist.ERA = Proj([pt-ssn], PATIENT)
Decllist.Id = ∅
Decllist.Vid = { (pt, [pt-ssn]) }
Decllist.Fid = ∅

Figure 4.15 shows the tables generated by the semantic functions for the declaration list "(pt : Patient)" at site A and site B. Note that the table at each site contains one row for each patient that appears in the database. These tables represent the relational algebra translations for the Decllist component of the query.

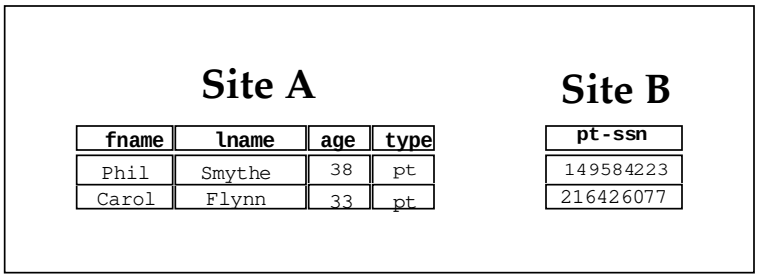


Figure 4.15: The relational algebra translations of the declaration list "(pt : Patient)" for the two relational databases shown in Figure 4.14.

4.3.3.2 The Predicate component

Recall from Section 4.3.1.1 that the relational translation of a predicate contains all bindings of the variables in the predicate for which the predicate is true. To compute this set, the semantic functions of the attribute grammar decompose predicates into their constituent syntactic parts, generate relational algebra expressions for each part, and subsequently combine the expressions such that only those bindings that satisfy the predicate appear in the result. The parse tree for the ReFER predicate "age-of(pt) > 55," which is shown in Figure 4.16, shows that the predicate is decomposed into a variable, a function, a comparison operator, and a constant.

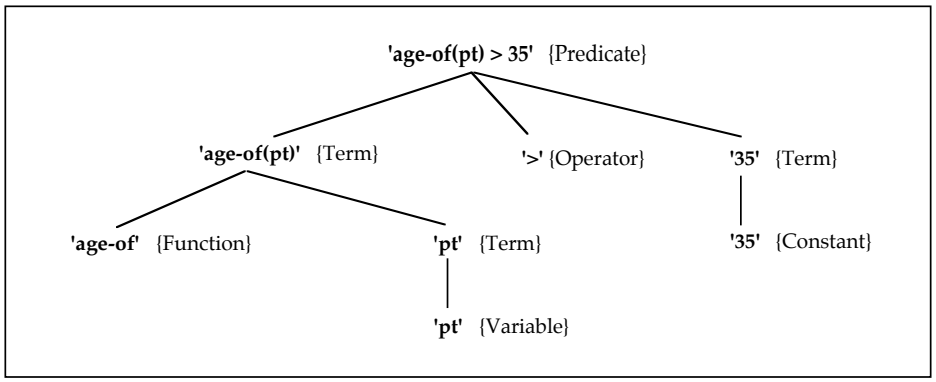


Figure 4.16: The parse tree for the Predicate component of the ReFER query. The context-free grammar decomposes the Predicate component into its constituent Function, Variable, Operator, and Constant components (labeled in braces).

The relational algebra expression for the variable "pt" is specified by the following semantic functions (recall that variables are, themselves, terms):

Grammar rule: Term ::= Var

Semantic functions:
Entity_Type = Lookup_Symbol_Table(Var.value)
Term.ERA = Entity_Type.ERA
Term.Id = Entity_Type.Id
Term.Vid = { (Variable.value, Entity_Type.Id) }
Term.Fid = Entity_Type.Fid

For each variable, the semantic functions generate a relational algebra expression that is equivalent to the mapping specified for the entity type over which the variable ranges. The first semantic function retrieves the identity of this entity type from the symbol table. The semantic functions generate a Vid value that associates the variable with the key fields of the relational algebra mapping for the entity type. Given these definitions and the mappings for the entity type Patient specified in Section 4.2, the semantic functions generate the following relational algebra translations of the term "pt" at site A and site B:

Site A

"pt" ::= Term.ERA = Sel([`type` = "`pt`"], `PERSON`)
Term.Id = ∅
Term.Vid = { (pt, [`lname`, `fname`]) }
Term.Fid = ∅

Site B

"pt" ::= Term.ERA = Proj([pt-ssn], PATIENT)
Term.Id = ∅
Term.Vid = { (pt, [pt-ssn]) }
Term.Fid = ∅

The relational algebra translation of a term consisting of a relationship function applied to a variable (such as "age-of(pt)") is specified by the following semantic functions:

Grammar rule: $Term_1$ ::= Function '(' $Term_2$ ')'

Semantic functions:
Relationship = Function.relationship
Source_Type = Function.source
Target_Type = Function.target
$Term_1$.ERA = Join([$Term_2$.Id = Relationship.Fid(Source_Type)], $Term_2$.ERA, Relationship.ERA)
$Term_1$.Id = Relationship.Fid(Target_Type)
$Term_1$.Vid = $Term_2$.Vid
$Term_1$.Fid = ∅

Recall that each relationship function of an FER schema is associated with an FER relationship type and maps from one FER entity type (the "source" entity type) to another FER entity type (the "target" entity type). For example, the relationship function "age-of" is associated with the relationship type "Person-Age," and maps from the source entity type "Person" to the target entity type "Integer" (see the FER schema in Figure 4.12). The first three semantic functions determine the relationship type, source entity type, and target entity type associated with a relationship function, as specified by the FER schema. The values of these parameters are substituted into the semantic functions that determine the ERA, Id, and Vid attribute values for the symbol "$Term_1$." The semantic functions evaluate references of the form "Relationship.Fid(<entity type>)" by returning the list of fields corresponding to <entity type> in the value of the Fid attribute for the relationship type. For example, the relational algebra mapping for the relationship type "Person-Age" at Site A is the following:

"Person-Age": Relationship.ERA = PERSON
Relationship.Id = ∅
Relationship.Vid = ∅
Relationship.Fid = { Person([lname, fname],
Integer([age]) }

Given this mapping, the value of Relationship.Fid(Person) is the field list [lname, fname] and the value of Relationship.Fid(Integer) is the field list [age].

Based on (1) the mappings specified in Section 4.2 for the relationship type Person-Age and (2) the relational algebra expressions previously computed for the term "pt," the semantic functions generate the following relational algebra translations of the term "age-of(pt)" at site A and site B:

Site A

"age-of(pt)" : Term.ERA = Join([lname = lname' and fname = fname'],
Sel([type = "pt"], PERSON),
PERSON')
Term.Id = [age']
Term.Vid = { (pt, [lname, fname]) }
Term.Fid = ∅

Site B

"age-of(pt)" : Term.ERA = Join([pt-ssn = pt-ssn],
Proj([pt-ssn], PATIENT),
Join([dob = date-of-birth],
Union(Proj([ssn, dob],
Rename([pt-ssn' => ssn],
PATIENT')),
Proj([ssn, dob],
Rename([md-ssn => ssn],
PHYSICIAN)),
DOB-AGE))
Term.Id = [age]
Term.Vid = { (pt, [pt-ssn]) }
Term.Fid = ∅

Note that when the same table appears twice in a relational algebra expression, the semantic functions implicitly rename one instance of the table (as well as any field names corresponding to that table); the renaming assures that references to separate instances of tables and fields are distinct (in the case of site A, renaming entails adding the prime symbol (') to the table name `PERSON` and to the field names `lname`, `fname`, and `age`). Figure 4.17 shows the tables generated by the semantic functions for the term "age-of(pt)" at site A and site B.

Site A

fname	lname	age	type	fname'	lname'	age'	type'
Phil	Smythe	38	pt	Phil	Smythe	38	pt
Carol	Flynn	33	pt	Carol	Flynn	33	pt

Site B

pt-ssn	ssn	dob	date-of-birth	age
149584223	149584223	12-2-57	12-2-57	38
216426077	216426077	3-25-62	3-25-62	33

Figure 4.17: The relational algebra translations of the term "age-of(pt)"for the two relational databases shown in Figure 4.14. The semantic functions generate these tables by first substituting all references to attributes with their appropriate values and subsequently evaluating the resulting relational algebra expressions.

The relational algebra expression for any constant that appears in a ReFER query is a *constant relation*, that is, a relation that consists of one field and one row [Maier 1983 p. 26]; the value of the field is the value of the constant. The name of the field is defined arbitrarily to be "constant-field." The following semantic functions generate a constant relation to represent the relational algebra translation of any ReFER term that is a constant:

Grammar rule: Term ::= Constant

Semantic functions: Term.ERA = Constant_relation(Constant.value)
Term.Id = [`constant-field`]
Term.Vid = ∅
Term.Fid = ∅

Because constant terms contain no variables, the value of the Vid attribute is NULL. Given this definition, the semantic functions generate the following relational algebra translation of the constant "35" at both site A and site B (the expressions are the same at both sites because no site-specific mappings are referenced):

"35": Term.ERA = Constant_relation("35")
Term.Id = [`constant-field`]
Term.Vid = ∅
Term.Fid = ∅

Constant relations may appear in relational algebra expressions anywhere that conventional tables may appear. This is an important property because constants may appear in ReFER queries anywhere that terms may appear. The relational algebra translations of constants, therefore, must be consistent with the translations for any other terms, such as variables or functions. For example, the constant "35" appears as one of the terms that is compared in the predicate "age-of(pt) > 35."

Recall that the semantic functions for translating any predicate that compares two terms, which I first presented in Section 4.3.2.2, specify a join of the relational algebra translations for each term (note that the relational algebra translation for the constant "35" must be a relational table for the JOIN operation to be a valid):

Grammar rule: Predicate ::= $Term_1$ Op $Term_2$

Semantic functions: Predicate.ERA = Join([$Term_1$.Id Op.value $Term_2$.Id],
$Term_1$.ERA,
$Term_2$.ERA)
Predicate.Id = ∅
Predicate.Vid = $Term_1$.Vid ∪ $Term_2$.Vid
Predicate.Fid = ∅

Given these semantic-function definitions and the relational algebra expressions generated for the terms "age-of(pt)" and "35," the semantic functions generate the following relational algebra translations for the predicate "age-of(pt) > 35" at site A and site B:

Site A

"age-of(pt) > 35" :

```
Term.ERA = Join( [ age' > constant-field ],
                 Join( [lname = lname' and fname = fname'],
                       Sel([type = "pt"], PERSON),
                       PERSON'),
                 Constant_relation("35"))
Term.Id = ∅
Term.Vid = { (pt, [lname, fname]) }
Term.Fid = ∅
```

Site B

"age-of(pt) > 35" :

```
Term.ERA = Join( [ age > constant-field ],
                 Join( [ pt-ssn = pt-ssn],
                       Proj( [pt-ssn], PATIENT),
                             Join([dob = date-of-birth],
                                  Union(Proj([ssn, dob],
                                             Rename([pt-ssn' => ssn],
                                                    PATIENT')),
                                        Proj([ssn, dob],
                                             Rename([md-ssn => ssn],
                                                    PHYSICIAN)),
                                  DOB-AGE)),
                 Constant_relation("35"))
Term.Id = [age]
Term.Vid = { (pt, [pt-ssn]) }
Term.Fid = ∅
```

Figure 4.18 shows the tables generated by the semantic functions for the predicate "age-of(pt) > 35" at site A and site B. Compare these tables to the tables in Figure 4.17 and note how the JOIN conditions specified in the semantic functions "filtered out" the bindings of the variable "pt" that did not satisfy the logic of the predicate (in Figures 4.17 and 4.18, the bindings of the variable "pt" are denoted by the values of the fields [lname, fname] at Site A and the value of the field [pt-ssn] at Site B). The resulting tables represent the relational algebra translations for the Predicate component of the ReFER query.

Site A

fname	lname	age	type	fname'	lname'	age'	type'	constant-field
Phil	Smythe	38	pt	Phil	Smythe	38	pt	35

Site B

pt-ssn	ssn	dob	date-of-birth	age	constant-field
149584223	149584223	12-2-57	12-2-57	38	35

Figure 4.18: The relational algebra translations for the predicate "age-of(pt) > 35" for the two relational databases shown in Figure 4.14.

4.3.3.3 The Termlist component

Recall from Section 4.3.1.1 that the semantic functions generate a relational algebra expression for the Termlist component that contains the values of the term list computed for all bindings of the variables that appear in the term list. The parse tree for the ReFER term list "last-name-of(pt)" is shown in Figure 4.19.

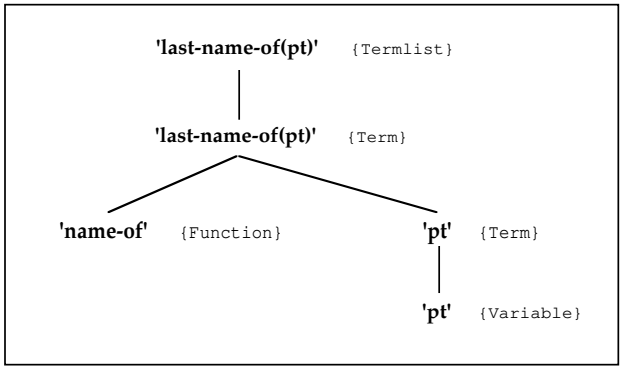


Figure 4.19: The parse tree for the Termlist component of a sample ReFER query. The context-free grammar decomposes the Termlist component into its constituent Function and Variable components (labeled in braces) . The parse tree indicates that the relational algebra translation of a term list that consists of only one term is the relational algebra translation of that term.

For term lists consisting of only one term, the relational algebra translation is equivalent to the relational algebra translation of that term. This equivalence is represented by the following semantic functions:

Grammar rule: Termlist ::= Term

Semantic functions: Termlist.ERA = Term.ERA
Termlist.Id = Term.Id
Termlist.Vid = Term.Vid
Termlist.Fid = Term.Fid

Given this equivalence, the following semantic functions are applied again to generate the appropriate relational translation of the term list "last-name-of(pt)":

Grammar rule: $Term_1$ ::= Function '(' $Term_2$ ')'

Semantic functions: Relationship = Function.relationship
Source_Type = Function.source
Target_Type = Function.target
$Term_1$.ERA = Join([$Term_2$.Id =
Relationship.Fid(Source_Type)],
$Term_2$.ERA,
Relationship.ERA)
$Term_1$.Id = Relationship.Fid(Target_Type)
$Term_1$.Vid = $Term_2$.Vid
$Term_1$.Fid = ∅

In this case, the relationship type associated with the function "last-name-of" is Person-Last_Name, the source entity type is Person, and the target entity type is String. Given these semantic-function definitions and the mappings for the relationship type Person-Last_Name at site A and site B, the semantic functions generate the following relational algebra translations for the term list "last-name-of(pt)":

Site A

"last-name-of(pt)" :
Term.ERA = Join([lname = lname' and fname = fname'],
Sel([type = "pt"], PERSON),
PERSON'),
Term.Id = [lname']
Term.Vid = { (pt, [lname, fname]) }
Term.Fid = ∅

Site B

"last-name-of(pt)" :

```
Term.ERA = Join( [ pt-ssn = ssn],
                 Proj( [pt-ssn], PATIENT),
                 Union(Proj([ssn, last],
                            Rename([pt-ssn' => ssn],
                                   PATIENT')),
                       Proj([ssn, last],
                            Rename([md-ssn => ssn],
                                   PHYSICIAN))))
Term.Id = [last]
Term.Vid = { (pt, [pt-ssn]) }
Term.Fid = ∅
```

Figure 4.20 shows the tables generated by the semantic functions for the term list "last-name-of(pt)" at site A and site B. These tables represent the relational algebra translations of the Termlist component of the query.

Site A

fname	lname	age	type	fname'	lname'	age'	type'
Phil	Smythe	38	pt	Phil	Smythe	38	pt
Carol	Flynn	33	pt	Carol	Flynn	33	pt

Site B

pt-ssn	ssn	last
149584223	149584223	Smythe
216426077	216426077	Flynn

Figure 4.20: The relational algebra translations of the term list"last-name-of(pt)" for the two relational databases shown in Figure 4.14.

4.3.3.4 The Range expression

Recall from Section 4.3.1.2 that the semantic functions of the attribute grammar join the expression generated for the Decllist component to the expression generated for the Predicate component to compute the set of bindings for which the Termlist component of a query should be evaluated. I defined the result of

this JOIN operation as the "Range" expression. The following semantic functions specify the operations required to generate the Range expression:

Grammar rule: Range ::= Decllist 'AND' Predicate

Semantic functions:
Range.ERA = Join([Decllist.Vid = Predicate.Vid],
Decllist.ERA,
Predicate.ERA)
Range.Id = ∅
Range.Vid = Decllist.Vid ∪ Predicate.Vid
Range.Fid = ∅

The semantic functions join the relational algebra translations of the Decllist and the Predicate components on the values of their respective Vid attributes; this is because the field lists denoted by the Vid values contain the variable bindings that are represented in each relational table. By only including in the result those variable bindings that appear in the relational algebra translation for the Predicate component, the semantic functions effectively "filter out" any bindings specified in the Decllist component that do not satisfy the logic of the predicate. These are precisely the semantics of the ReFER query language for the Range component. Given these semantic-function definitions and the relational algebra expressions generated for the Decllist and Predicate components, the semantic functions defined for the Range component generate the following relational algebra translations for "(pt : Patient) AND age-of(pt) > 35" at site A and site B:

Site A

"(pt : Patient) AND age-of(pt) > 35":

```
Term.ERA = Join( [ lname" = lname and fname" = fname],
                 Sel([type" = "pt"], PERSON")
                 Join( [ age'  > constant-field ],
                       Join( [lname = lname' and fname = fname' ],
                             Sel([type = "pt"], PERSON),
                             PERSON' ),
                       Constant_relation("35")))
Term.Id = ∅
Term.Vid = { (pt, [lname, fname]) }
Term.Fid = ∅
```

Site B

"(pt : Patient) AND age-of(pt) > 35":

```
Term.ERA = Join( [pt-ssn" = pt-ssn],
                 Proj( [pt-ssn"], PATIENT")
                 Join( [ age' > constant-field ],
                       Join( [ pt-ssn = pt-ssn' ],
                             Proj( [pt-ssn], PATIENT),
                             Join([dob = date-of-birth],
                                  Union(Proj([ssn,dob],
                                             Rename([pt-ssn' => ssn],
                                                    PATIENT')),
                                        Proj([ssn,dob],
                                             Rename([md-ssn => ssn],
                                                    PHYSICIAN)),
                                  DOB-AGE)),
                       Constant_relation("35")))
Term.Id = ∅
Term.Vid = { (pt, [pt-ssn]) }
Term.Fid = ∅
```

Note the renaming of certain tables and fields with the double-prime symbol (") in these relational algebra expressions; the renaming is performed again to distinguish separate instances of identically named tables and fields. Figure 4.21 shows the tables generated by the semantic functions for the Range expression "(pt : Patient) AND age-of(pt) > 35" at site A and site B. Compare these tables to the tables in Figure 4.15 and note how the JOIN conditions specified in the semantic functions "filtered out" the bindings of the variable "pt" that did not satisfy the logic of the predicate (in Figures 4.15 and 4.21, the bindings of the variable "pt" are denoted by the values of the fields [lname, fname] at Site A and the value of the field [pt-ssn] at Site B). The resulting tables represent the relational algebra translations of the Range component of the query.

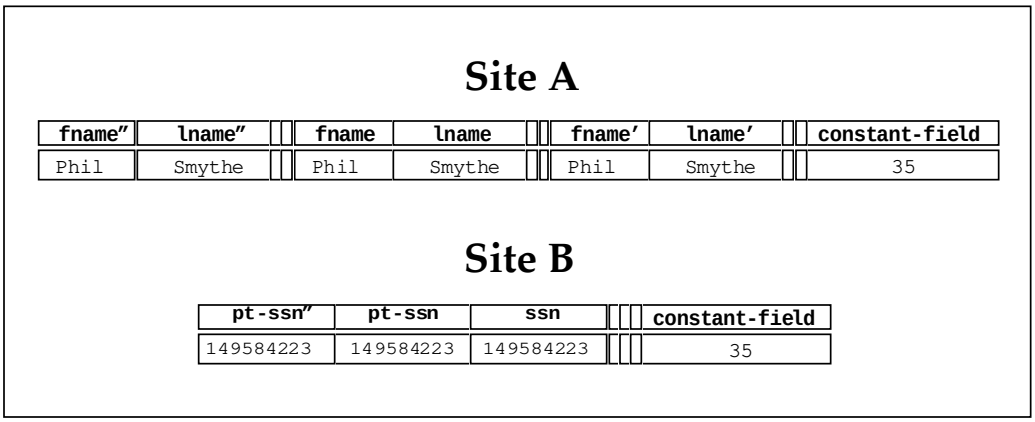


Figure 4.21: The relational algebra translations of the range expression "(pt : Patient) AND age-of(pt) > 35" for the two relational databases shown in Figure 4.14. Note that I have "condensed" the representation of certain fields in the tables to save space

4.3.3.5 The complete query

The semantic functions of the attribute grammar generate the complete relational algebra translation for a ReFER query by joining the expression generated for the Range component to the expression generated for the Termlist component, and subsequently projecting the values of the Termlist component. The following semantic functions perform this operation:

Grammar rule: Query ::= '[' Termlist ' | ' Range ']'

Semantic functions:
```
Query.ERA = Proj( [Termlist.Id],
                  Join([ Termlist.Vid = Range.Vid ],
                       Termlist.ERA,
                       Range.ERA)
Query.Id = Termlist.Id
Query.Vid = Termlist.Vid
Query.Fid = Termlist.Fid
```

Again, the semantic functions join the Termlist and Range expressions on their respective Vid values to "filter out" the bindings of the variable "pt" in the relational translation for the Termlist component that do not satisfy the logic of the predicate. Because the complete relational algebra expressions generated by these semantic functions are extensive, I leave their formulation as an exercise. However, Figure 4.22 shows the tables that result from evaluating the relational

algebra translations produced at site A and site B. These tables represent the site-specific relational translations of the sample ReFER query.

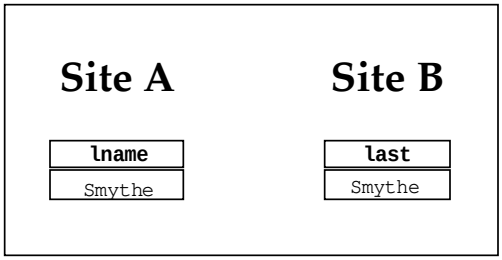


Figure 4.22: The results of evaluating the relational algebra translations of the query "last-name-of(pt) | (pt : Patient) AND age-of(pt) > 35" for the two relational databases shown in Figure 4.14.

Most relational DBMSs cannot evaluate relational algebra expressions directly, however, because commercial DBMSs only offer query interfaces based on SQL (a relational calculus language). At each site, therefore, TransFER must convert the relational algebra translations produced by the attribute grammar to equivalent SQL queries that can be executed by the local DBMS.

4.3.4 Translating relational algebra to SQL

The translation of ERA expressions to SQL queries is a straightforward process, owing to the semantic equivalence of a relational algebra augmented with aggregation and the relational calculus [Klug 1982]. Several algorithms for performing this translation exist [Klug 1982, Ullman 1988]. The only complication is that certain subexpressions of a relational algebra query must be formulated as *separate SQL queries*, rather than as SQL subqueries. Specifically, relational algebra subexpressions that specify the UNION operation or the AGGREGATE operation must be formulated as distinct SQL queries. The reason is that the current SQL standard does not allow SQL union or aggregation operations to appear within the relational expressions in the "FROM" list of an SQL query. For example, the following SQL query is not valid:

```
SELECT  last
FROM  (SELECT pt-ssn, last FROM PATIENT)
          UNION
        (SELECT md-ssn, last FROM PHYSICIAN).
```

Therefore, the results of SQL union and aggregation operations must be saved in temporary intermediate tables; the names of these tables appear in the FROM lists of subsequent SQL queries. The results of relational algebra subexpressions that specify *external functions* also must be computed separately and stored in temporary tables. The reason is that external functions, by definition, cannot be executed by an SQL query engine. The SQL query engine can retrieve only the arguments to external functions; the functions must subsequently operate on these arguments in the host environment and return their results to temporary tables in the database. These tables then appear in subsequent SQL queries. Therefore, a single relational algebra expression that results from the attribute-grammar translation of a ReFER query may be translated into a sequence of multiple SQL queries that communicate via intermediate results saved in temporary tables. The final result of a ReFER query translation is a *database program* consisting of (potentially) multiple SQL queries interleaved with run-time commands to create temporary tables and to call external functions. Figure 4.24 shows the site-specific output of a TransFER query translation.

```
Begin database program {

INSERT INTO Temp_Table (Field1, Field2)
  ( SELECT pt-ssn, dob
    FROM PATIENT )
  UNION
  ( SELECT md-ssn, dob
    FROM PHYSICIAN)


SELECT pt.lname
FROM Temp_Table tt, PATIENT pt
WHERE (datediff(year,tt.Field2,getdate()) > 35 AND
      (pt.pt-ssn = tt.Field1)

} End database program
```

operations may, in fact, be *redundant* in the sense that if the operations were omitted, the meanings of the relational algebra expressions would be unchanged. Redundant relational algebra operations are introduced into query translations by the way in which the TransFER attribute grammar combines the site-specific mappings for FER entity types and relationship types when translating ReFER queries. Although redundant operations do not affect the correctness of the relational algebra expressions generated by the attribute grammar, they do, in practice, increase the time and computing resources that are required to evaluate the expressions. For this reason, it is desirable to detect and to remove redundant relational operations from expressions produced by the attribute grammar. I now describe the theory and the mechanism for performing certain of these optimizations.

The TransFER optimizer detects and eliminates redundant JOIN operations by applying a theory of "PROJECT equivalence" that is based on the relational algebra. In this section, I describe the theory, I prove its correctness, and I illustrate its application to the problem of optimizing relational algebra expressions produced by the TransFER attribute grammar.

4.4.1 The theory of PROJECT equivalence

The TransFER compiler applies optimizations that are based on a specific kind of equivalence between relational algebra expressions called "PROJECT equivalence." Informally, PROJECT equivalence between two relational algebra expressions (that is, relations) means that the two relations are identical except for the existence of certain *duplicated attributes*. A duplicated attribute is an attribute of a relation, the value of which is equal to the value of another attribute for every row of the relation. For example, two "PROJECT-equivalent" relations appear in Figure 4.23. PROJECT equivalence between two relations implies that the same PROJECT operation performed over both relations produces identical results.

The expression "$R \triangleq S$" means that the relations R and S are PROJECT equivalent. I define PROJECT equivalence formally as follows:

> **Definition**: If $R \triangleq S$, then there exists some transformation Δ (defined below) such that the application of Δ to R generates S, that is, $\Delta(R) = S$.

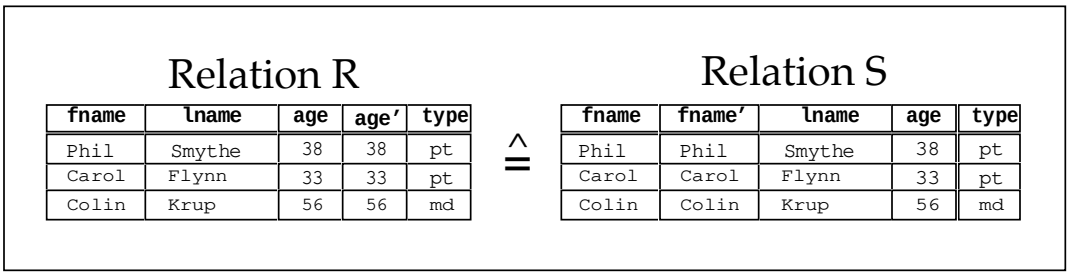
The transformation Δ consists of some finite sequence of transformation functions δ (defined below) such that $\Delta = [\delta_1,\delta_2,..,\delta_n]$ where $n \geq 0$. Each transformation function δ must consist of one of the following operations:

1. *Duplicate*. For some attribute α of the relation R, a new attribute α' is added to the relation R such that α' has the same domain as α and $\alpha'=\alpha$ in every row of R. The Duplicate operation essentially reproduces one column of the relation R and assigns a unique attribute name to the new column. Because the new column contains the same values as an existing column, the Duplicate operation adds no information.

2. *Merge*. For some pair of attributes α, α' of the relation R where $\alpha = \alpha'$ in every row of R, the attribute α' is eliminated from the relation R. The merge function essentially eliminates a redundant column in the relation R, generating a relation with only one "copy" of the column. Because the eliminated column contained the same values as an existing column, the Merge operation removes no information.

The sequential application of the transformation functions $[\delta_1,..,\delta_n]$ to the relation R performs the transformation $\Delta(R)$, such that if $R \triangleq S$ and $\Delta = [\delta_1,\delta_2,..,\delta_n]$, then $\Delta(R) = \delta_n(..\delta_2(\delta_1(R))..) = S$. Note that Duplicate and Merge are inverse operations. This property implies that if a transformation Δ exists such that $\Delta(R) = S$, then some transformation Δ' exists such that $\Delta'(S) = R$. Also note that two relations are PROJECT equivalent if and only if they have the same cardinality, because neither the Duplicate nor Merge operations can change the cardinality of a relation.

Figure 4.23 shows two relations that are PROJECT equivalent. The relation R can be transformed to the relation S by first duplicating the attribute `fname` in R and subsequently merging the attributes `age` and `age'`. The inverse sequence of Duplicate and Merge operations perform the transformation from S to R.



Relation R

fname	lname	age	age'	type
Phil	Smythe	38	38	pt
Carol	Flynn	33	33	pt
Colin	Krup	56	56	md

$\hat{=}$

Relation S

fname	fname'	lname	age	type
Phil	Phil	Smythe	38	pt
Carol	Carol	Flynn	33	pt
Colin	Colin	Krup	56	md

Figure 4.23: Two relations that are PROJECT equivalent. A sequence of one Merge and one Duplicate operation can transform either relation into the other.

PROJECT equivalence is relevant to query optimization because the same projection of two PROJECT-equivalent relational algebra expressions produces the same result. This implies that if the elimination of a JOIN from a relational algebra expression generates a PROJECT-equivalent expression, then any subsequent projection of that expression will produce the same results as if the JOIN had not been eliminated. This property allows the optimizer to remove certain JOINs without changing the meaning of the query. Formally, I define this property of PROJECT equivalence as Theorem 4.1. Unless otherwise stated, I will use "conventional" relational algebra notation in the remainder of this chapter. The uppercase letters R, S, and T will denote relations; uppercase letters near the beginning and end of the alphabet will denote sets of relational attributes (for example, A, B, X, Y); the symbol α will denote a single relational attribute.

Theorem 4.1 if $R \hat{=} S$,

then $\pi_X R = \pi_{\rho_{R \to S}(X)} S$

The function $\rho_{R \to S}(X)$ is a *renaming* operation that renames certain attributes in the attribute list X. The renaming takes place so that the names of the projected attributes are consistent with the transformation between R and S that is defined by $R \hat{=} S$. Specifically, the renaming operation $\rho_{R \to S}$ defines a *sequence* of renaming functions $[\rho_1, \rho_2, .., \rho_n]$ such that each ρ_i corresponds to one of the transformation functions $\delta_i \in \Delta$ where $\Delta(R) = S$. Each ρ_i consists of an ordered pair of attribute names $<\alpha \to \alpha'>$ such that ρ_i renames every instance of the attribute name α in an attribute list to the attribute name α'. The pair of attribute names for each ρ_i that corresponds to a δ_i is determined as follows:

1. If δ_i is a Merge operation that merges the attributes α and α' such that only the attribute α remains, then the pair of attribute names for ρ_i is $<\alpha' \rightarrow \alpha>$.

2. If δ_i is a Duplicate operation that duplicates the values of the attribute α, then the pair of attribute names for ρ_i is $<\alpha \rightarrow \alpha>$ (that is, no renaming is specified).

Informally, the renaming operation ensures that any attribute α' in the relation R that is eliminated by a Merge operation when R is transformed to S is appropriately replaced by the attribute α in S if α' appears in the list of attributes to be projected. Because the Duplicate operation does not eliminate any attributes, no renaming is required. For example, the following renaming operation is defined by the transformation from the relation R to the relation S in Figure 4.23:

$$\rho_{R \rightarrow S} = [<\texttt{fname} \rightarrow \texttt{fname}>, <\texttt{age}' \rightarrow \texttt{age}>].$$

When applied to the attribute list [fname, lname, age'], the renaming operation produces the attribute list [fname, lname, age]. Inspection of the relations shown in Figure 4.23 demonstrates that this operation is required for PROJECT equivalence to hold in the following example:

$$\pi_{[\text{fname,lname,age}']}R = \pi_{[\text{fname,lname,age}]}S.$$

Theorem 4.1 is proved by induction on the sequential application of the transformation functions $\delta_1,..,\delta_n$ that constitute the transformation Δ between R and S. I claim (without proof) that Theorem 4.1 holds equally whether I use the PROJECT or the MULTIPROJECT operation in the following proof (I therefore use the symbol π to denote either operation):

> *Proof*: The inductive hypothesis is that Theorem 4.1 holds before any transformations are performed, that is, when R = S. In this case, $\Delta = \varnothing$, which implies that $\rho_{R \rightarrow S} = \varnothing$ by the definition of the renaming operation. If $\rho_{R \rightarrow S} = \varnothing$, then $\rho_{R \rightarrow S}(X) = X$. This equality proves the inductive hypothesis because we know that $\pi_X R = \pi_X S$ when R = S. The inductive step is that Theorem 4.1 holds when a single transformation

function, δ, is applied to the relation R to produce the relation S. There are two cases:

Case 1. δ is a Duplicate operation that adds the attribute α′ to R, which contains the same values as the existing attribute α for every row. Because the Duplicate operation creates a NULL renaming operation (that is, $\rho_{R \to S} = [<A \to A>]$), we know that $\rho_{R \to S}(X) = X$ for any attribute list X. Therefore, the theorem holds if the equality $\pi_X R = \pi_X S$ holds given the Duplicate operation that transforms R to S. Because the relation S is simply the relation R with one additional attribute (α′), I can write $R = \pi_Y S$ where Y is the original set of attributes in the relation R. Substituting, the theorem holds if the equality $\pi_X(\pi_Y S) = \pi_X S$ holds. Because Y is the original set of attributes of R, we know that $X \subseteq Y$, because otherwise the expression $\pi_X R$ would be invalid. This containment implies, by an equivalence of the relational algebra [Ullman 1989], (p. 664) that $\pi_X(\pi_Y S) = \pi_X S$.

Case 2. δ is a Merge operation that eliminates the duplicate attribute α' from R, where α′ contains the same values for every row as some remaining attribute α. In this case, $\rho_{R \to S} = \{<\alpha' \to \alpha>\}$, so that each instance of α′ in the attribute list X is replaced by α. Two subcases exist:

2a. X contains no instances of α′. In this case, $\rho_{R \to S}(X) = X$ and the theorem holds if the equality $\pi_X R = \pi_X S$ holds for the Merge operation that transforms R to S. Because the relation S is simply the relation R with one attribute removed (α′), I can write $S = \pi_Z R$, where Z is the original set of attributes in R minus the removed attribute α′. Substituting, the theorem holds if the equality $\pi_X R = \pi_X(\pi_Z R)$ holds. Because we have assumed in this case that X does not contain the removed attribute, α′, we know that $X \subseteq Z$. Therefore, by the same equivalence, $\pi_X R = \pi_X(\pi_Z R)$.

2b. X contains one or more instances of α′. In this case, each such instance is renamed to α. By the definition of the

Merge operation, we know that the value of α′ in each row of R is equal to the value of α in each row of S. We also know that the values of the attributes in the attribute list X-α' (that is, the attributes that are common to both relations) are equal in each row of R and S because the Merge operation did not affect these attributes. Therefore, projecting the combination of the values of α′ and X-α′ in the relation R produces the same result as projecting the value of α and X-α′ from the relation S. In other words, $\pi_{\chi}R = \pi_{\rho_{R\rightarrow S}(X)}S$. □

4.4.2 An application of PROJECT equivalence

In Section 4.3, I showed how the semantic functions of the TransFER attribute grammar may introduce redundant operations when translating ReFER queries to relational algebra expressions. For example, the semantic functions generate the following relational translation for the term "last-name-of(pt)" at site B:

```
Term.ERA = Join( [ pt-ssn = ssn],                          (expression 1)
                 Proj( [pt-ssn], PATIENT),
                 Union(Proj([ssn, last],
                            Rename([pt-ssn' => ssn],
                                   PATIENT')),
                       Proj([ssn, last],
                            Rename([md-ssn => ssn],
                                   PHYSICIAN)))).
```

Inspection of the relational schema for the database at site B (shown in Figure 3.15) reveals that the last name of each patient can be computed by the much simpler relational algebra expression:

```
Term.ERA = Proj( [ pt-ssn, last], PATIENT).                (expression 2)
```

The semantic functions of the attribute grammar produce the more complex relational algebra translation for "last-name-of(pt)" because the site-specific and site-independent semantic functions of the attribute grammar must be specified

in the most general way possible. For example, the site-specific mapping for the relationship type Person-Last_Name must specify a union of the PATIENT and the PHYSICIAN tables, because the relationship type may be used to derive the last names of patients, physicians, or both (recall that Person-Last_Name is defined for the supertype Person in the FER schema). When the relationship type is used to derive the last name of a patient, as in "last-name-of(pt)," no data from the PHYSICIAN table is needed and the union is unnecessary. At the time the mapping is specified, however, there is no way to anticipate how it will be applied in any particular query, so it must be specified to accommodate the most general case. Likewise, the semantic function that translates the application of a relationship function (such as "last-name-of") to a term (such as "pt") must specify a JOIN between the relational algebra translation of the relationship function and the relational algebra translation of the term. The JOIN is necessary because the semantic function is *site-independent* and, in the most general case, a JOIN is required to derive the associations between entities. When the relationship function "last-name-of" is used to derive the last name of a patient at site B, however, the JOIN is unnecessary because the last name is represented in the same table as the patient. The role of the TransFER optimizer is to recognize these unnecessary operations in specific relational algebra expressions and to transform the expressions into equivalent expressions without the operations. The optimizer exploits the properties of PROJECT equivalence to perform certain of these transformations.

For example, the following relational algebra equivalence allows the optimizer to transform the original expression produced by the attribute grammar for the term "last-name-of(pt)" (expression 1) to the simpler expression that I produced manually (expression 2) without changing the meaning of the expression (again, I use the "conventional" relational algebra notation):

Theorem 4.2 $$(\pi_X R \cup \pi_Y S) \underset{A = B}{\bowtie} \pi_Z R \triangleq \pi_{(X \cup Z)} R$$

if the list of attributes A = the list of attributes B, and the values of the attributes in A are unique in the relation $(\pi_X R \cup \pi_Y S)$.

The optimizer eliminates one JOIN in transforming the original expression, which has the form of the left-hand side of the equivalence in Theorem 4.2, to the simpler expression, which has the form of the right-hand side. The optimizer can apply this transformation because the last relational algebra operation specified by the semantic functions of the attribute grammar is a multiprojection (see Section 4.3.1.2). If the attributes that are projected by this operation are appropriately renamed, then the properties of PROJECT equivalence ensure that the results of the query will be the same after the transformation is performed. However, for Theorem 4.2 to hold, the PROJECT operations denoted on the left-hand and right-hand sides of the PROJECT equivalence must be "classic" PROJECT operations, that is, PROJECT operations that eliminate duplicate rows. The reason for this constraint is apparent in the proof of Theorem 4.2 that appears in the next section.

4.4.3 Proving correctness of the optimizations

The proof of Theorem 4.2 proceeds in two steps. The first step proves the relational equivalence:

Lemma 4.1 $$(\pi_X R \cup \pi_Y S) \underset{A = B}{\bowtie} \pi_Z R = \pi_X R \underset{A = B}{\bowtie} \pi_Z R$$

if the list of attributes A = the list of attributes B,
and the values of the attributes in A are unique in
the relation $(\pi_X R \cup \pi_Y S)$

Note that the symbol "=" appears in Lemma 4.1, rather than "$\triangleq$." The second step proves a more general PROJECT equivalence, which actually is the basis for a large number of optimizations:

Theorem 4.3 $$\pi_X R \underset{A = B}{\bowtie} \pi_Z R \triangleq \pi_{(X \cup Z)} R$$

if the list of attributes A = the list of attributes B,
and the values of the attributes in A are unique in
either the relation $\pi_X R$ or $\pi_Z R$

Note that Theorem 4.3 may be applied to the proof of Theorem 4.2 because the condition that the values of the attributes in A are unique in the relational expression ($\pi_X R \cup \pi_Y S$) *implies* that the values of the attributes in A are unique in the relational expression $\pi_X R$. The combination of Lemma 4.1 and Theorem 4.3 proves Theorem 4.2 because of the following properties of PROJECT equivalence:

1. Relational equivalence implies PROJECT equivalence, that is, if R = S, then $R \triangleq S$. The definition of PROJECT equivalence specifies that one possible transformation, Δ, from R to S such that $R \triangleq S$ is the NULL transformation, that is, $\Delta = \varnothing$. Thus, if R = S, then $\Delta(R) = S$ where $\Delta = \varnothing$ and $R \triangleq S$. This property implies the following:

   if $\quad (\pi_X R \cup \pi_Y S) \underset{A = B}{\bowtie} \pi_Z R = \pi_X R \underset{A = B}{\bowtie} \pi_Z R$

   then $\quad (\pi_X R \cup \pi_Y S) \underset{A = B}{\bowtie} \pi_Z R \triangleq \pi_X R \underset{A = B}{\bowtie} \pi_Z R$.

2. PROJECT equivalence is transitive, that is, if $R \triangleq S$ and $S \triangleq T$, then $R \triangleq T$. Transitivity holds because one can simply *compose* the sequence of transformation functions that transform R to S and S to T to transform R to T, that is, if $\Delta_1(R) = S$ where $\Delta_1 = \{\delta_1,..,\delta_n\}$ and $\Delta_2(S) = T$ where $\Delta_2 = \{\delta_{n+1},..,\delta_m\}$, then $\Delta_3(R) = T$ where $\Delta_3 = \Delta_1 + \Delta_2 = \{\delta_1,..,\delta_n,\delta_{n+1},..,\delta_m\}$. The transitivity of PROJECT equivalence implies the following:

   if $\quad (\pi_X R \cup \pi_Y S) \underset{A = B}{\bowtie} \pi_Z R \triangleq \pi_X R \underset{A = B}{\bowtie} \pi_Z R$ $\quad$ and

   $$\pi_X R \underset{A = B}{\bowtie} \pi_Z R \triangleq \pi_{(X \cup Z)} R$$

   then $\quad (\pi_X R \cup \pi_Y S) \underset{A = B}{\bowtie} \pi_Z R \triangleq \pi_{(X \cup Z)} R.$

These two properties confirm that the proof of Theorem 4.2 rests on the proofs of Lemma 4.1 and Theorem 4.3.

Proof of Lemma 4.1: By the distributive property of the JOIN operator in relational algebra,

$$(\pi_X R \cup \pi_Y S) \underset{A=B}{\bowtie} \pi_Z R = (\pi_X R \underset{A=B}{\bowtie} \pi_Z R) \cup (\pi_Y S \underset{A=B}{\bowtie} \pi_Z R).$$

The proof of Lemma 4.1, therefore, depends on establishing the following equivalence:

$$(\pi_X R \underset{A=B}{\bowtie} \pi_Z R) \cup (\pi_Y S \underset{A=B}{\bowtie} \pi_Z R) = (\pi_X R \underset{A=B}{\bowtie} \pi_Z R)$$

I prove this equivalence by considering three possibilities for the value of the attribute A: The value appears only in the relation $\pi_X R$, the value appears only in the relation $\pi_Y S$, and the value appears in both the relations $\pi_X R$ and $\pi_Y S$.

Case 1. The value of the attribute A appears only in $\pi_X R$ and not in $\pi_Y S$. For all such values,

$$(\pi_Y S \underset{A=B}{\bowtie} \pi_Z R) = \varnothing$$

because the JOIN condition cannot be satisfied by any values of A in $\pi_Y S$. This implies

$$(\pi_X R \underset{A=B}{\bowtie} \pi_Z R) \cup (\pi_Y S \underset{A=B}{\bowtie} \pi_Z R) = (\pi_X R \underset{A=B}{\bowtie} \pi_Z R) \cup \varnothing$$
$$= (\pi_X R \underset{A=B}{\bowtie} \pi_Z R)$$

Case 2. The value of the attribute A appears only in $\pi_Y S$ and not in $\pi_X R$. For all such values,

$$(\pi_Y S \underset{A=B}{\bowtie} \pi_Z R) = \varnothing.$$

This is true because both the set of attributes X and the set of attributes Z must contain the set of attributes A; if they do not, the original expression

$$(\pi_X R \cup \pi_Y S) \underset{A = B}{\bowtie} \pi_Z R$$

is not a valid relational algebra expression (recall that the set of attributes A = the set of attributes B). Given that both X and Z contain A, if the values of the attributes A do not appear in $\pi_X R$, then they cannot appear in $\pi_Z R$ (because both expressions include the projection of the attributes A). Therefore, for all values of the attributes A that do not appear in $\pi_X R$,

$$(\pi_Y S \underset{A = B}{\bowtie} \pi_Z R) = \varnothing$$

because the JOIN condition cannot be satisfied by any values of A in $\pi_Z R$. This implies

$$(\pi_X R \underset{A = B}{\bowtie} \pi_Z R) \cup (\pi_Y S \underset{A = B}{\bowtie} \pi_Z R) = (\pi_X R \underset{A = B}{\bowtie} \pi_Z R) \cup \varnothing$$

$$= (\pi_X R \underset{A = B}{\bowtie} \pi_Z R).$$

Case 3. The value of the attribute A appears in *both* $\pi_X R$ and in $\pi_Y S$. Because we know that the values of A are unique in $(\pi_X R \cup \pi_Y S)$, this implies that the values of each attribute in the attribute set X-A and in the attribute set Y-A *are the same* for each value in the columns of A that appears in both $\pi_X R$ and $\pi_Y S$, respectively; the values of X-A and of Y-A must be the same so that the relational UNION operator will "merge" the rows from $\pi_X R$ and $\pi_Y S$ that have the same values of A into a single row when the expression $(\pi_X R \cup \pi_Y S)$ is evaluated. Therefore, for each pair of rows in $\pi_X R$ and $\pi_Y S$ that have the same values of the attributes A, the values of the other attributes are also the same. In other words, the rows are identical, that is $\pi_X R = \pi_Y S$. For all such rows,

$$(\pi_X R \underset{A = B}{\bowtie} \pi_Z R) = (\pi_Y S \underset{A = B}{\bowtie} \pi_Z R) = T.$$

Substituting,

$$(\pi_X R \underset{A=B}{\bowtie} \pi_Z R) \cup (\pi_Y S \underset{A=B}{\bowtie} \pi_Z R) = T \cup T = T$$

$$= (\pi_X R \underset{A=B}{\bowtie} \pi_Z R).$$

Because I have shown that the equivalence

$$(\pi_X R \underset{A=B}{\bowtie} \pi_Z R) \cup (\pi_Y S \underset{A=B}{\bowtie} \pi_Z R) = (\pi_X R \underset{A=B}{\bowtie} \pi_Z R)$$

holds for all of the possible values of the attributes in A, I have proven Lemma 4.1.□

Given Lemma 4.1, Theorem 4.2 is proved by proving Theorem 4.3:

$$\pi_X R \underset{A=B}{\bowtie} \pi_Z R \triangleq \pi_{(X \cup Z)} R$$

if the list of attributes A = the list of attributes B,
and the values of the attributes in A are unique in
either the relational expression $\pi_X R$ or $\pi_Z R$.

Proof: Assume that the values of the attributes in A are unique in $\pi_X R$ (a symmetrical proof exists for the case in which the values are unique in $\pi_Z R$). There are three possible *containment relationships* between the sets of attributes X and Z: $X = Z$ (implying that $X \cup Z = X$), $X \supset Z$ (implying that $X \cup Z = X$), and $X \subset Z$ (implying that $X \cup Z = Z$). I prove Theorem 4.3 by proving that the following assertions hold for each containment relationship:

1. The expression on the left-hand side of the PROJECT equivalence *has the same cardinality* as the expression on the right-hand side.

2. There exists a *sequence of Merge operations* that transforms the expression on the left-hand side of the PROJECT equivalence to the expression on the right-hand side.

By the definition of PROJECT equivalence, these two assertions prove Theorem 4.3. In the proof below, I adopt the following notational convention: When the same relation appears twice in an expression, I rename one "copy" of the relation

and each attribute of that relation with the prime (′) symbol. For example, this renaming changes the expression

$$\pi_X R \underset{A = A}{\bowtie} \pi_Y R$$

(where A is a set of attributes in the relation R) to the expression

$$\pi_X R \underset{A = A'}{\bowtie} \pi_Y R'.$$

If A = the set of attributes $\{\alpha_1,..,\alpha_n\}$ in the relation R, then A′ = $\{\alpha_1', ..,\alpha_n'\}$ in the relation R′. This convention helps to distinguish copies of attributes with the same name that result from joining a relation to itself.

Case 1. X = Z. In this case, I must show

$$\pi_X R \underset{A = A'}{\bowtie} \pi_{X'} R' = T \triangleq \pi_X R.$$

Because A is unique in $\pi_X R$, A′ is unique in $\pi_{X'} R'$. This implies a one-to-one relationship between the rows of $\pi_X R$ and $\pi_{X'} R'$ for each value of the attributes in A; therefore, $|T| = |\pi_X R|$, by the definition of the JOIN operation. For each attribute $\alpha' \in X'$, either $\alpha' \in A'$ or $\alpha' \in X'\text{-}A'$ (note that A′ must be a subset of X′ for the JOIN operation to be valid). For each $\alpha' \in A'$, there is an $\alpha \in A$ such that (1) both α′ and α are in the relation T (because $A' \subset X'$ and $A \subset X$) and (2) in each row of T, the value of α′ = the value of α (because that equality is implied by the JOIN condition). Therefore, for each $\alpha' \in A'$, the attributes α′ and α in T may be merged into the single attribute α, by the definition of the Merge operation. For each $\alpha' \in X'\text{-}A'$, there must be a functional dependency $A' \rightarrow \alpha'$ in R′, such that there is only one possible value of α′ for each value of A′; this dependency must exist for the values of A′ to be unique in $\pi_{X'} R'$ because otherwise there could be two values of α′ for each value of A′ and A′ would not be unique in $\pi_{X'} R'$. By the same argument, there is an identical functional dependency $A \rightarrow \alpha$ in R because the values of A are unique in $\pi_X R$. Because the value of A′ = the value of A in each row of T (by the JOIN condition), the functional

dependencies imply that the value of α' = the value of α in each row of T. Therefore, for each $\alpha' \in X' - A'$, the attributes α' and α in T may be merged into the single attribute α. Because every $\alpha' \in X'$ may be merged with an α in the relation T (that is, in the result of the JOIN), there exists a transformation from T to $\pi_X R$ that is consistent with the definition of PROJECT equivalence; therefore, T and $\pi_X R$ are PROJECT equivalent.

Case 2. $X \supset Z$. In this case, I must show

$$\pi_X R \underset{A = A'}{\bowtie} \pi_Z R' = T \triangleq \pi_X R.$$

Because the values of A are unique in $\pi_X R$, the values of A′ must be unique in $\pi_Z R'$ because $X \supset Z$ (a more restrictive projection certainly cannot create any duplicate values). Therefore, there is a one-to-one relationship between the rows of $\pi_X R$ and $\pi_Z R'$ for each value of the attributes in A, implying again that $|T| = |\pi_X R|$. Note that for each attribute $\alpha' \in Z$, there must be a corresponding $\alpha \in X$ because $X \supset Z$. For each $\alpha' \in A'$, there is an $\alpha \in A$ such that (1) both α' and α are in the relation T and (2) in each row of T, the value of α' = the value of α (because that equality is implied by the JOIN condition). Therefore, for each $\alpha' \in A'$, the attributes α' and α in T may be merged into the single attribute α. For each $\alpha' \in Z - A'$, there must be a functional dependency $A' \rightarrow \alpha'$ in R′ (by the same argument as in Case 1) because A′ is unique in $\pi_Z R'$. Given $A' \rightarrow \alpha'$ in R′, there is an identical functional dependency $A \rightarrow \alpha$ in R (recall that R and R′ are the same relations). Because the value of A′ = the value of A in each row of T (by the JOIN condition), the functional dependencies imply that the value of α' = the value of α in each row of T. Therefore, for each $\alpha' \in Z - A'$, the attributes α' and α in T may be merged into the single attribute α. Because every $\alpha' \in Z$ may be merged with an α in the relation T (that is, in the result of the JOIN), there exists a transformation from T to $\pi_X R$ that is consistent with the definition of PROJECT equivalence, again proving that T and $\pi_X R$ are PROJECT equivalent.

Case 3. $X \subset Z$. In this case, I must show

$$\pi_X R \underset{A = A'}{\bowtie} \pi_Z R' = T \triangleq \pi_Z R'.$$

Because A is unique in $\pi_X R$, there is exactly one row in $\pi_X R$ that corresponds to each row in $\pi_Z R'$ such that the value of A = the value of A' (that is, such that the JOIN condition holds). This implies a many-to-one relationship between the rows of $\pi_Z R'$ and $\pi_X R$ for each value of A' (note that the values of A' are not necessarily unique in $\pi_Z R'$). Therefore, $|T| = |\pi_Z R'|$, by the definition of the JOIN operation. Note that for each attribute $\alpha \in X$, there must be a corresponding $\alpha' \in Z$ because $X \subset Z$. For each $\alpha \in A$, there is an $\alpha' \in A'$ such that (1) both α and α' are in the relation T and (2) in each row of T, the value of α = the value of α' (because that equality is implied by the JOIN condition). Therefore, for each $\alpha \in A$, the attributes α and α' in T may be merged into the single attribute α'. For each $\alpha \in X\text{-}A$, there must be a functional dependency $A \rightarrow \alpha$ in R, because A is unique in $\pi_X R$ (see Case 1). Given $A \rightarrow \alpha$ in R, there is an identical functional dependency $A' \rightarrow \alpha'$ in R' (recall that R and R' are the same relations). Because the value of A = the value of A' in each row of T (by the JOIN condition), the functional dependencies imply that the value of α = the value of α' in each row of T. Therefore, for each $\alpha \in X\text{-}A$, the attributes α and α' in T may be merged into the single attribute α'. Because every $\alpha \in X$ may be merged with an α' in the relation T (that is, in the result of the JOIN), there exists a transformation from T to $\pi_Z R'$ that is consistent with the definition of PROJECT equivalence; therefore, T and $\pi_Z R$ are PROJECT equivalent. □

The proof of Theorem 4.3 completes the proof of Theorem 4.1. However, the TransFER optimizer applies Theorem 4.3 to justify a large set of simplifying transformations. For example, Theorem 4.3 implies each of the following PROJECT equivalences:

Corollary 4.3a $\quad R \underset{A = A}{\bowtie} R \triangleq R$

Corollary 4.3b $\pi_X R \underset{A = A}{\bowtie} R \triangleq R$

Corollary 4.3c $\pi_X R \underset{A = A}{\bowtie} \pi_Y R \triangleq \pi_X R$ if $X \supseteq Y$

Lastly, one easily can generalize Theorem 4.3 to Theorem 4.4:

Theorem 4.4 $(\pi_X R \underset{C = D \quad A = B}{\bowtie} S) \underset{E = F}{\bowtie} (\pi_Z R \underset{}{\bowtie} T) \triangleq ((\pi_{(X \cup Z)} R \underset{C = D}{\bowtie} S) \underset{E = F}{\bowtie} T)$

if the list of attributes A = the list of attributes B,
the attributes A appear only in the relation $\pi_X R$, the
attributes B appear only in the relation $\pi_Z R$, and
the values of the attributes A are unique in *either*
the relation $\pi_X R$ or $\pi_Z R$

The proof of Theorem 4.4 follows directly from the commutativity and associativity properties of the JOIN operator [Ullman 1988, p. 62] and is sufficiently straightforward that I have omitted it.

4.4.4 The optimization theorems in practice

Theorem 4.3 and its corollaries are useful for eliminating many JOIN operations that appear in the relational translations of ReFER queries. The wide applicability of Theorem 4.3 is based on the way in which the site-specific semantic functions of the attribute grammar specify entity-type and relationship-type mappings and the way in which the site-independent semantic functions of the attribute grammar translate relationship functions and relationship predicates.

Recall that the site-specific mapping for each entity type specifies (1) a relational algebra expression in which each row represents a distinct instance of the entity type and (2) a set of key fields that represent the identifier of each instance of the entity type (see Section 4.2.1). The semantic functions that specify these mappings assign the relational algebra expression to the attribute ERA and the set of key fields to the attribute Id. Because each row in the relational algebra expression represents a distinct entity and the values of the key fields represent

the identifier of each entity, the values of the key fields are unique in the relational algebra expression. For example, the following semantic functions specify the mapping for the entity type Person at Site A (see Figure 4.11 for the complete schema at Site A):

Person : Entity_Type.ERA = "PERSON"
Entity_Type.Id = [lname, fname]

This mapping implies that the concatenated values of the fields [lname, fname] are unique in the table PERSON at site B. Therefore, by Theorem 4.3, if the mapping for the entity type Person is joined to another instance of the table PERSON on the fields [lname, fname], that JOIN may be eliminated without changing the meaning of the relational algebra query. Note that such a JOIN will occur in the translation of "last-name-of(p)" (where "p" is declared to range over the entity type Person) at site A because the site-specific mapping for the relationship type Person-Last_Name also assigns the table PERSON to the ERA attribute:

Person-Last_Name: Relationship_Type.ERA = "PERSON"
Relationship_Type.Fid = { Person([lname, fname]),
String([lname]) }

The semantic functions for the mappings of both Person and Person-Last_Name assign the table PERSON to the ERA attribute because at Site A, the representation of each person's last name is stored as a *field* in the same table that stores the set of Person entities (see Figure 4.11). Given the definition of the site-independent semantic function that translates terms that apply relationship functions (such as "last-name-of"), the relational translation of the term "last-name-of(p)" at Site A is

Term.ERA = Join([lname = lname' and fname = fname'],
PERSON,
PERSON'),
Term.Id = [lname']
Term.Vid = { (pt, [lname, fname]) }
Term.Fid = ∅.

Note that the value of the ERA attribute takes the form

$$R \underset{A = A'}{\bowtie} R'$$

where R and R′ both represent the table `PERSON` and A and A′ both represent the attribute list [`lname, fname`]. By Corollary 4.3a, the relational translation of the term "last-name-of(p)" at Site A may be simplified to

Term.ERA = `PERSON`
Term.Id = [`lname`]
Term.Vid = { (pt, [`lname, fname`]) }
Term.Fid = ∅

provided that all subsequent references to `lname'` and `fname'` are replaced by references to `lname` and `fname`. The corollary of Theorem 4.3 may be applied because the values of the attribute list [`lname, fname`] are unique in the table `PERSON`, as Theorem 4.3 requires.

4.5 Summary

The TransFER query-translation methodology provides a principled approach to translating abstract clinical queries to equivalent and site-specific relational database programs. Based on the theories of relational algebra and attribute-grammar translation, the translation methodology supports (1) the formal specification of database mappings to bridge representational heterogeneity, (2) the systematic composition of site-specific mappings to translate arbitrary queries, and (3) the principled optimization of relational algebra translations to improve computational efficiency. However, the ultimate utility of the TransFER methodology can be judged only in the context of the problem that TransFER addresses: providing a uniform query interface to disparately implemented clinical databases. In the next chapter, I present the criteria that I used to evaluate TransFER's utility and I describe the design and the results of a formal evaluation that I conducted to test TransFER's utility.