A Systematic Method for Query Evaluation in Federated Relational Databases

Yangjun Chen and Wolfgang Benn
Department of Computer Science, Technical University of Chemnitz-Zwickau
09107 Chemnitz, Germany

Abstract. In this paper, a systematic method for evaluating queries issued to an federated relational database system is presented. The method consists of four phases: syntactic analysis, query decomposition, query translation and result synthesis, which all can be done automatically based on the metadata built in the system, even if the structure conflicts among the local databases exist.

1. Introduction

With the advent of applications involving increased cooperation among systems, the development of methods for integrating pre-existing databases becomes important. The design of such global database systems must allow unified access to the diverse and possibly heterogeneous database systems without subjecting them to conversion or major modifications [BOT86, LA86, Jo90, SK92, CW93, HLM94, RPRG94, KFMRN96]. One important issue in such a system is the query treatment, which has received much attention during the past several years. In [NTA96], a query processing method is proposed based on the concept of "virtual function dependency" to derive reasonable values for *null values* which occur due to the integration. In [LP96], a query decomposition strategy has been developed based on a simple one-to-one concept mapping. Several earlier papers such as [SC94, LHS95, LHSC95] belong to special case studies. No systematic method has been suggested at all. Especially, in the case of structure conflicts, no idea on it emerges. (Although in [LW96] such problems are discussed, the method proposed there, based on the concept of *superrelations*, can not be automated.) The other researches reported in [LOG93, DSD94, DSD95, DS96, CB96, ETB96] are mainly concerned with optimization.

In this paper, we address this problem and try to develop a systematic method for the (select-project-join) query evaluation in a (relational) heterogeneous environment. Our method consists of four steps: syntactic analysis, query decomposition, query translation and result synthesis. If the medadata are well defined, all of them can be performed automatically. First, the query decomposition can be done by using correspondence assertions. Secondly, the query translation can be automated based on the relation structure terms and the corresponding derivation rules, which are higher order logic concepts for accommodating complicated semantic heterogeneities. The last step: result synthesis is merely a simple process, by which the local answers are combined together.

The remainder of the paper is organized as follows. In Section 2, we show our system architecture and the data dictionary used for storing meta information.Then, in Section 3, the query treatment is discussed in detail. Finally, we set forth a short summary in Section 4.

2. System architecture and metadata

Before we discuss the main idea of our method for evaluating queries submitted to a federated

The work reported here is supported by DFG (Deutsche Forschungsgemeinschaft) under grant No. Be1786/1-1.

system, we simply describe our system architecture and the metadata used to resolve the semantic conflicts among the component databases.

2.1 System architecture

Our system architecture consists of three-layers: FSM-client, FSM and FSM-agents as shown in Fig. 1 (here FSM represents “Federated System Manager”.)

The task of the FSM-client layer consists in the application management, providing a suite of application tools which enable users and DBAs to access the system. The FSM layer is responsible for the mergence of potentially conflicting local databases and the definition of global schemas. In addition, a centralized management for the data dictionary (DD) is supported at this layer. The FSM-agent layer corresponds to the local system management, addressing all the issues w.r.t. schema translation and export as well as local transaction and query processing.

In term of this architecture, each component database is installed in some FSM-agent and must be registered in the FSM. Then, for a component relational database, each attribute value will be implicitly prefixed with a string of the form:

<FSM-agent name>.<database system name>.<database name>.<relation name>.<attribute name>,

where “.” denotes string concatenation. For example, *FSM_agent1.informix.PatientDB.patient_records.name* references attribute “name” from relation “patient_records” in an informix database named “PatientDB”, installed in “FSM_agent1”.

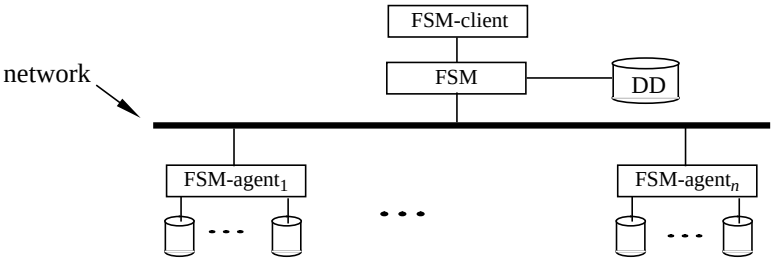

Fig. 1. System architecture

2.2 Metadata classification

In this section, we discuss the meta information built in our system, which can be classified into three groups: structure mappings, concept mappings and data mappings, each for a different kind of semantic conflicts: structure conflict, concept conflict and data conflict.

2.2.1 Structure mappings

In the case of relational databases, we consider three kinds of structure conflicts which can be illustrated as shown in Fig. 2.

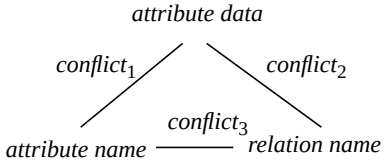

Fig. 2 Illustration for structure conflicts

They are,

1) when an attribute value in one database appears as an attribute name in another database,

2) when an attribute value in one database appears as a relation name in another database,

3) when an attribute name in one database appears as a relation name in another database.

As an example, consider three local schemas of the following form:.

DB_1: faculty(name, research_area, income),

DB_2: research(research_area, $name_1$, ..., $name_n$),

DB_3: $name_1$'(research_area, income),
... ...
$name_m$'(research_area, income).

In DB_1, there is one single relation, with one tuple per faculty member and research area, storing his/her income. In DB_2, there is one single relation, with one tuple per research area, and one attribute per faculty member, named by his/her name and storing its income. Finally, DB_3 has one relation per faculty member, named by his/her name; each relation has one tuple per research area storing the income.

If we want to integrate these three databases and the global schema R is chosen to be the same as "faculty", then an algebra expression like $\pi_{\text{name, research\_area}}(\sigma_{\text{income}>1000}(R))$ has to be translated so that it can be evaluated against different local schemas. For example, in order to evaluate this expression against DB_3, it should be converted into the following form:

for each $y \in \{name_1', name_2', ..., name_m'\}$ **do**
$\{\pi_{\text{research\_area}}(\sigma_{\text{income}>1000}(y))\}$.

A translation like this is needed when a user of one of these databases wants to work with the other databases, too.

In order to represent such conflicts formally and accordingly to support an automatic transformation of queries in case any of such conflicts exist, we introduce the concept of *relation structure terms* (RST) to capture higher-order information w.r.t. a local database. Then, for the RSTs w.r.t. some heterogeneous databases, we define a set of derivation rules to specify the semantic conflicts among them.

Relation structure terms

In our system, an RST is defined as follows:

$$[re_{\{R1, ..., Rm\}} | a_1: x_1, a_2: x_2, ..., a_l: x_l, y: z_{\{A1, ..., An\}}],$$

where *re* is a variable ranging over the relation name set $\{R_1, ..., R_m\}$, y is a variable ranging over the attribute name set $\{A_1, ..., A_n\}$, $x_1, ..., x_l$ and z are variables ranging over respective attribute values, and $a_1, ..., a_l$ are attribute names. In the above term, each pair of the form: a_i: x_i (i = 1, ..., l) or y: z is called an attribute descriptor. Obviously, such an RST can be used to represent either a collection of relations possessing the same structure, or part structure of a re-

lation. For example, [$re_{\{\text{name1'}, ..., \text{name}m'\}}$| research_area: x, income: y] represents any relation in DB_3, while an RST of the form: [$re_{\{\text{research}\}}$| research_area: x, y: $z_{\{\text{name1}, ..., \text{name}n\}}$] (or simply ["research"| research_area: x, y: $z_{\{\text{name1}, ..., \text{name}n\}}$]) represents a part structure of "research" with the form: research(research_area,..., name_i, ...) in DB_2. Since such a structure allows variables for relation names and attribute names, it can be regarded as a higher order predicate quantifying both data and metadata. When the variables (of an RST) appearing in the relation name position and attribute name positions are all instantiated to constants, it is degenerated to a first-order predicate. For example, ["faculty" | name: x_1, research_area: x_2, income: x_3] is a first-order predicate quantifying tuples of R_1.

The purpose of RSTs is to formalize both data and metadata. Therefore, it can be used to declare schematic discrepancies. In fact, by combining a set of RSTs into a derivation rule, we can specify some semantic correspondences of heterogeneous local databases exactly.

For convenience, an RST can be simply written as [re | a_1: x_1, a_2: x_2, ..., a_l: x_l, y: z] if the possible confusion can be avoided by the context.

Derivation rules

For the RSTs, we can define derivation rules in a standard way, as implicitly universally quantified statements of the form: γ_1 & γ_2 ... & $\gamma_l \Leftarrow \tau_1$ & τ_2 ... & τ_k, where both γ_i's and τ_k's are (partly) instantiated RSTs or normal predicates of the first-order logic. For example, using the following two rules

$r_{\text{DB1-DB3}}$: [y |research_area: x, income: z] $\Leftarrow$
["faculty" |name: y, research_area: x, income: z], $y \in$ {name_1', name_2', ..., name_m'},
$r_{\text{DB3-DB1}}$: ["faculty"|name: x, research_area: y, income: z] $\Leftarrow$
[x |research_area: y, income: z], $x \in$ {name_1", name_2", ..., name_l"},

the semantic correspondence between DB_1 and DB_3 can be specified. (Note that in $r_{\text{DB3-DB1}}$, name_1", name_2", ..., and name_l" are the attribute values of "name" in "faculty".)

Similarly, using the following rules, we can establish the semantic relationship between DB_1 and DB_2:

$r_{\text{DB1-DB2}}$: ["research" |research_area: y, x: z] $\Leftarrow$
["faculty"|name: x, research_area: y, income: z], $x \in$ {name_1, name_2, ..., name_n},
$r_{\text{DB2-DB1}}$: ["faculty" |name: x, research_area: y, income: z] $\Leftarrow$
["research" |research_area: y, x: z], $x \in$ {name_1", name_2", ..., name_l"}.

Finally, in a similar way, the semantic correspondence between DB_2 and DB_3 can be constructed as follows:

$r_{\text{DB3-DB2}}$: ["research" |research_area: x, y: z] $\Leftarrow$
[y |research_area: x, income: z], $y \in$ {name_1, name_2, ..., name_n},
$r_{\text{DB2-DB3}}$: [y |research_area: x, income: z] $\Leftarrow$
["research" |research_area: x, y: z], $y \in$ {name_1', name_2', ..., name_m'}.

In the remainder of the paper, a conjunction consisting of RSTs and normal first-order predicates is called a c-expression (standing for "complex expression"). For a derivation rule of the form: $A \Leftarrow B$, A and B are called the antecedent part and the consequent part of the rule, respec-

tively.

2.2.2 Concept mappings

The second semantic conflict is concerned with the concept aspects, caused by the different perceptions of the same real world entities.

[SP94, SPD92] proposed simple and uniform correspondence assertions for the declaration of semantic, descriptive, structural, naming and data correspondences and conflicts (see also [Du94]). These assertions allow to declare how the schemas are related, but not to declare how to integrate them. Concretely, four semantic correspondences between two concepts are defined in [SP94], based on their *real-world states* (*RWS*). They are equivalence (≡), inclusion (⊇ or ⊆), disjunction (∅) and intersection (∩). Equivalence between two concepts means that their extensions (populations) hold the same number of occurrences and that we should be able to relate those occurrences in some way (e.g., with their *object identifiers*). Borrowing the terminology from [SP94], a correspondence assertion can be informally described as follows:

$S_1 \bullet A \equiv S_2 \bullet B$, iff $RWS(A) = RWS(B)$ always holds,
$S_1 \bullet A \subseteq S_2 \bullet B$, iff $RWS(A) \subseteq RWS(B)$ always holds,
$S_1 \bullet A \cap S_2 \bullet B$, iff $RWS(A) \cap RWS(B) \neq \phi$ holds sometimes,
$S_1 \bullet A \varnothing S_2 \bullet B$, iff $RWS(A) \cap RWS(B) = \phi$ always holds.

For example, assuming *person*, *book*, *faculty* and *man* are four concepts (relation or attribute names) from S_1 and *human*, *publication*, *student*, and *woman* are another four concepts from S_2, the following four assertions can be established to declare their semantic correspondences, respectively: $S_1 \bullet person \equiv S_2 \bullet human$, $S_1 \bullet book \subseteq S_2 \bullet publication$, $S_1 \bullet faculty \cap S_2 \bullet student$, $S_1 \bullet man \varnothing S_2 \bullet woman$.

Experience shows that only the above four assertions are not powerful enough to specify all the semantic relationships of local databases. Therefore, an extra assertion: derivation (→) has to be introduced to capture more semantic conflicts, which can be informally described as follows. The derivation from a set of concepts (say, $A_1, A_2, ..., A_n$) to another concept (say, B) means that each occurrence of B can be derived by some operations over a combination of occurrences of $A_1, A_2, ...,$ and A_n, denoted $A_1, A_2, ..., A_n \rightarrow B$. In the case that $A_1, A_2, ...,$ and A_n are from a schema S_1 and B from another schema S_2, the derivation is expressed by $S_1(A_1, A_2, ..., A_n) \rightarrow S_2 \bullet B$, stating that $RWS(A_1, A_2, ..., A_n) \rightarrow RWS(B)$ holds at any time. For example, a derivation assertion of the form: $S_1(parent, brother) \rightarrow S_2 \bullet uncle$ can specify the semantic relationship between *parent* and *brother* in S_1 and *uncle* in S_2 clearly, which can not be established otherwise.

2.2.3 Data mappings

As to the data mappings, there are different kinds of correspondences that must be considered.

1) (exact correspondence) In this case, a value in one database corresponds to at most one value in another database. Then, we can simply make a binary table for such pairs.

2) (function correspondence) This case is similar to the first one. The only difference being that a simple function can be used to declare the relevant relation. For example, consider an attribute “height_in_inches” from one database and an attribute “height_in_centimeters” from another. The value correspondence of these two attributes can be constructed by defining a

function of the form:

$$y = f(x) = 2.54 \cdot x,$$

where y is a variable ranging over the domain of "height_in_inches" and x is a variable ranging over "height_in_centimeters". Further, a fact of the form: S_1•height-in-inches ≡ S_2•height-in-centimeters should be declared to indicate that both of them refer to the same concept of the *real-world*.

3) (fuzzy correspondence) The third case is called the fuzzy correspondence, in which a value in one database may corresponds to more than one value in another database. In this case, we use the fuzzy theory to describe the corresponding semantic relationship [BCG96]. For example, consider two attributes "age_1" and "age_2" from two different databases, respectively. If the value set of "age_1" A is {1, 2, ..., 100} while the value set of "age_2" B is {infantile, child, young, adult, old, very_old}, then the mapping from "age_1" to "age_2" may be of the following form:

{(1, infantile, 1), (2, infantile, 0.9), ...,
(3, child, 1), ..., (13, child, 1), ...,
(14, young, 0.5), (15, young, 0.6), ..., (20, young, 1), ...},

in which each (a, b) with $a \in A$ and $b \in B$ is associated with a value $v \in [0, 1]$ to indicate the degree to which a is relevant to b.

2.2.4 Meta information storage

All the above meta information are stored in the data dictionary and accommodated into a *part-of* hierarchy of the form as shown in Fig. 3.

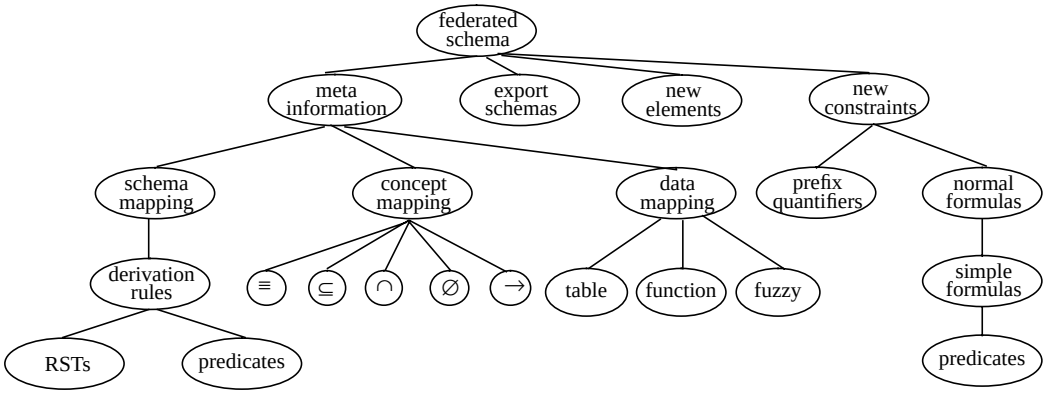

Fig. 3 Data Dictionary

The intention of such an organization is straightforward. First, in our opinion, a federated schema is mainly composed of two parts: the export schemas and the associated meta information, possibly augmented with some new elements. Accordingly, classes "*export schemas*" and "*meta information*" are connected with class "*federated schema*" using part-of links (see Fig. 3). In addition, two classes "*new elements*" and "*new constraints*" may be linked in the case that some new elements are generated for the integrated schema and some new semantic constraints must be made to declare the semantic relationships between the participating local databases. It should be noticed that in our system, for the two local databases considered, we always take

one of them as the basic integrated version, with some new elements added if necessary. For example, if S_1•*person* ≡ S_2•*human* is given, we may take *person* as an element (as a relation name or an attribute name) of the integrated schema. (But for evaluating a query concerning *person* against the integrated schema, both S_1•*person* and S_2•*human* need to be considered.) However, if S_1•*faculty* ∩ S_2•*student* is given, some new elements such as $IS_{faculty,\ student}$, $IS_{faculty-}$, $IS_{student-}$ and *student* will be added into S_1 if we take S_1 as the basic integrated schema, where $IS_{faculty,\ student} = S_1$•*faculty* ∩ S_2•*student*, $IS_{faculty-} = S_1$•*faculty* ∩ ¬$IS_{faculty,\ student}$ and $IS_{student-} = S_2$•*student* ∩ ¬$IS_{faculty,\ student}$. On the other hand, all the integrity constraints appearing in the local databases are regarded as part of the integrated schema. But some new integrity constraints may be required to construct the semantic relationships between the local databases. As an example, consider a database containing a relation *Department*(*name, emp*, ...) and another one containing a relation *Employee*(*name, dept*, ...), a constraint of the form: ∀*e*(in *Employee*)∀*d*(in *Department*)(*d.name* = *e.Dept* → *e.name* in *d.emp*) may be generated for the integrated schema, representing that if someone works in a department, then this department will have him/her recorded in the *emp* attribute. Therefore, the corresponding classes should be predefined and linked according to their semantics (see below for a detailed discussion).

Furthermore, in view of the discussion above, the meta information associated with a federated schema can be divided into three groups: structure mappings, concept mappings and data mappings. Each structure mapping consists of a set of derivation rules and each rule is composed of several RSTs and predicates connected with "," (representing a *conjunction*) and "⇐". Then, the corresponding classes are linked in such a way that the above semantics is implicitly implemented. Meanwhile, two classes can be defined for RSTs and predicates, respectively. Further, as to the concept mappings, we define five subclasses for them with each for an assertion. At last, three subclasses named "*table*", "*function*" and "*fuzzy*" are needed, each behaving as a "subset" of class "*data mapping*".

In the following discussion, ***C*** represents the set of all classes and the type of a class $C \in \boldsymbol{C}$, denoted by *type*(*C*), is defined as:

$type(C) = \langle a_1{:}type_1, ..., a_l{:}type_l, Agg_1$ with cc_1: $out\text{-}type_1, ..., Agg_k$ with cc_k: $out\text{-}type_k, m_1, ..., m_h\rangle$,

where a_i represents an attribute name, Agg_j represents an aggregation function: $C \rightarrow C'$ ($C, C' \in \boldsymbol{C}$ and $out\text{-}type_j \in type(\boldsymbol{C})$), m_g stands for a method defined on the object identifiers or on the attribute values of objects and $type_i$ is defined as follows:

$type_i$ ::= <PrimitiveTyp>|<list>|<set>|<ClassType>,
<PrimitiveTyp> ::= <Integer> | <Boolean> | <Character> | <String>|<Real>,
<list> ::= "[" $type_i^+$ "]",
<set> ::= "{" $type_i^+$ "}".

Furthermore, each aggregation function may be associated with a cardinality constraint $cc_j \in$ {[1:1], [1:*n*], [*m*:1], [*m*:*n*]} (j = 1, ..., k).

Then, in our implementation, we have

type("*federated schema*") = <*IS*: <*string*>, S_f: <*string*>, S_s: <*string*>, *indicator*: <*boolean*>, Agg_1 with [1:1]: <*type*("*meta information*")>,

Agg_2 with [1:2]: <*type*(“*export schemas*”)>,
Agg_3 with [1:1]: <*type*(“*new elements*”)>,
Agg_4 with [1:1]: <*type*(“*new constraints*”)>>,

where *IS* stands for the integrated schema name, S_f and S_s for the two participating local schemas’, *indicator* is used to indicate whether S_f or S_s is taken as the basic integrated version and each Agg_j is an aggregation function, through which the corresponding objects of the classes connected with “*federated schema*” using *part-of* links can be referenced.

As an example, an object of this class may be of the form: oid_1(*IS*: IS_DB, S_f: S_1, S_s: S_2, *indicator*: 0, ...), representing an integration process as illustrated in Fig. 4(a), where S_1 is used as the basic integrated schema, since the value of *indicator* is 0. Otherwise, if the value of *indicator* is 1, S_2 will be taken as the basic integrated schema.

(a) (b)

Fig. 4. Integration process

With another object, say oid_2(*IS*: IS_DB’, S_f: IS_DB, S_s: S_3, ...) together, a more complicated integration process as shown in Fig. 4(b) can be represented.

Class “*export schemas*” has a relatively simple structure as follows:

type(“*export schemas*”) = <*S*: <*string*>, *path*: <*concatenation of strings*>, *r_a_names*: <*set of pairs*>>,

where *S* is an attribute for the storage of a local database name, *path* is for the access path of a database in the FSM system, denoted as given in 2.1 and *r_a_names* is for an export schema, stored as a set of pairs of the form: (*r_name*, {$attr_1$, ..., $attr_n$}). Here, *r_name* is a relation name and each $attr_i$ is an exported attribute name.

The type of “*meta information*” is defined as follows:

type(“*meta information*”) = <S_f_S_s: <*pairs of strings*>,
Agg_1 with [1:*n*]: <*type*(“*structure mapping*”)>,
Agg_2 with [1:*n*]: <*type*(“*concept mapping*”)>,
$Agg_3$with [1:*n*]: <*type*(“*data mapping*”)>>,

where S_f_S_s is used to store the pair of local database names, for which the meta information is constructed, while Agg_1, Agg_2 and Agg_3 are three aggregation functions, through which the objects of classes “*structure mapping*”, “*concept mapping*” and “*data mapping*” can be referenced, respectively.

As discussed above, any new element is defined by some function over the existing local elements (such as $IS_{faculty\text{-}} = S_1 \bullet faculty) \cap \neg IS_{faculty,\ student}$.) Then, a set of functions has to be defined in “*new elements*”. In general, class “*new elements*” has the following structure:

type(“*new elements*”) = <*S*: <*string*>, *new_elem*: <*set*>, m_1, ..., m_h>.

Here, *S* stands for the name of a new element added to the integrated schema, *new_elem* is for the attributes of the new element, stored as a set and each element in it is itself a set of the form: $\{a, a_1, ..., a_n, m_i\}$, where *a* represents the new attribute, each a_j is a local attribute and m_i is a method name defined over $a_1, ..., a_n$.

Example 1. To illustrate class "*new elements*", let us see one of its objects, which may be of the form:

$$\text{oid}(S\text{: } IS_{faculty,\ student},\ new\_elem\text{: } \{\{name,\ S_1 \bullet faculty \bullet name,\ S_2 \bullet student \bullet name,\ m\},\ \{income,\ S_1 \bullet faculty \bullet income,\ S_2 \bullet student \bullet study\_support,\ m'\}\}),$$

where $S_1 \bullet faculty \bullet name$ and $S_1 \bullet faculty \bullet income$ stand for two attributes of S_1, while $S_2 \bullet student \bullet name$ *and* $S_2 \bullet student \bullet study\_support$ are two attribute names of S_2, *m* is a method name, implementing the following function:

$$f(x, y) = \begin{cases} x & \text{if there exist tuple } t_1 \in faculty \text{ and tuple } t_2 \in student \text{ such that } t_1.name = x, \\ & t_2.name = y \text{ and } x = y \text{ (in terms of data mappping)}, \\ Null & \text{otherwise.} \end{cases}$$

and *m*' is another one for the function below:

$$g(x, y) = \begin{cases} \frac{x+y}{2} & \text{if there exist tuple } t_1 \in faculty \text{ and tuple } t_2 \in student \text{ such that } t_1.name = t_2.name \\ & \text{(in terms of data mappping), and } x = t_1.income \text{ and } y = t_2.study\_support; \\ Null & \text{otherwise.} \end{cases}$$

Then, this object represents a new relation (named $IS_{faculty,\ student}$) with two attributes: "*name*" and "*income*". The first attribute corresponds to the attribute "*name*" of *faculty* in S_1 (through method *m*) and the second is defined using *m*'.

Example 2. As another example, assume that the relation schemas of *faculty* and *student* are *faculty*(*name*, *income*, *research_area*) and *student*(*name*, *study_support*), respectively. In this case, we may not create new elements for "*research_area*". But if we want to do so, a new attribute can be defined as follows:

$$\{work\_area,\ S_1 \bullet faculty \bullet research\_area,\ \_,\ m\},$$

where *m* represents a function of the following form:

$$h(x, \_) = \begin{cases} x & \text{if there exist tuple } t_1 \in faculty \text{ and tuple } t_2 \in student \text{ such that } t_1.name = t_2.name \\ & \text{and } t_1.research\_area = x, \\ Null & \text{otherwise.} \end{cases}$$

Conversely, if the relation schemas of *faculty* and *student* are *faculty*(*name*, *income*) and *student*(*name*, *study_support*, *study_area*), respectively, we define a new attribute as follows:

$$\{work\_area,\ S_2 \bullet student \bullet study\_area,\ \_,\ m'\},$$

where *m*' represents a function of the following form:

$$r(\_, y) = \begin{cases} y & \text{if there exist tuple } t_1 \in faculty \text{ and tuple } t_2 \in student \text{ such that } t_1.name = t_2.name \\ & \text{and } t_2.study\_area = y, \\ Null & \text{otherwise.} \end{cases}$$

At last, if the relation schemas of *faculty* and *student* are *faculty*(*name*, *income*, *research_area*)

and *student*(*name, study_support, study_area*), respectively, the method associated with the new attribute can be defined as follows:

$$\{work\_area, S_1 \bullet faculty \bullet research\_area, S_2 \bullet student \bullet study\_area, \_, m''\},$$

where m'' is a method name for the following function:

$$u(x, y) = \{x\} \cup \{y\}.$$

In our system, each new integrity constraint is of the following form:

$$(Qx_1 \in T_1) \ldots (Qx_n \in T_n) e(x_1, \ldots, x_n),$$

where Q is either $\forall$ or $\exists$, $n > 0$, *exp* is a (quantifier-free) boolean expression (concretely, two normal formulas connected with "$\rightarrow$", each of them is of the form: $(p_{11} \vee \ldots p_{1n_1}) \wedge \ldots \wedge (p_{j1} \vee \ldots p_{jn_j})$), $x_1, \ldots, x_n$ are all variables occurring in *exp*, and $T_1, \ldots, T_n$ are set-valued expressions (or class names). For example, Therefore, two classes "*prefix quantifier*" and "*normal formulas*" are defined as parts of "*new constraints*" (see Fig. 3). Then, class "*new constraints*" is of the following form:

type("*new constraints*") = <*constraint_number*: <*string*>,
Agg_1 with [1:1]: <*type*("*prefix quantifier*")>,
Agg_2 with [1:2]: <*type*("*normal formulas*")>>,

where *constraint_number* is used to identify an newly generated individual integrity constraint and Agg_1 and Agg_2 are two aggregation functions, through which the objects of classes "*prefix quantifier*" and "*normal formulas*" can be referenced, respectively. Accordingly, "*prefix quantifier*" is of the form:

type("*prefix quantifier*") = <*constraint_number*: <*string*>, *quantifiers*: <string>>,

and "*normal formulas*" is of the form:

type("*normal formulas*") = <*constraint_number*: <*string*>,
l_formula with [1:*n*]: <*type*("*formulas*")>,
r_formula with [1:*n*]: <*type*("*formulas*")>>,

where *quantifiers* is a single-valued attribute used to store a string of the form: $(Qx_1 \in T_1) \ldots (Qx_n \in T_n)$, while *l_formula* and *r_formula* are two attributes to store the left and right hand sides of "$\rightarrow$" in an expression, respectively.

Similarly, we can define all the other classes shown in Fig. 3 in such a way that the relevant information can be stored. However, a detailed description will be tedious but without difficulty, since all the mapping information are well defined in 2.2 and the corresponding data structures for them can be determined easily. Therefore, we omit them for simplicity. In the following, we mainly discuss a query treatment technique based on the meta information stored in the data dictionary.

3. Query treatment

Based on the metadata built as above, a query submitted to an integrated schema can be evalu-

ated in a four-phase method (see Fig. 5).

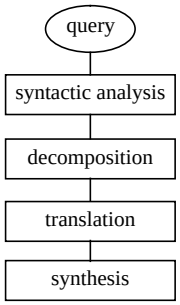

Fig 5. Query treatment

First, the query will be analyzed syntactically (using *LEX* unix utility [Ra87]). Then, it will be decomposed in terms of the correspondence assertions. Next, we translate any decomposed subquery in terms of the derivation rules so that it can be evaluated in the corresponding component database. At last, a synthesis process is needed to combine the local results evaluated. In the following, we discuss the last three issues in 3.1, 3.2 and 3.3, respectively.

3.1 Query decomposition

We consider the select-project-join queries of the following form:

$$\pi_{A_1 \ldots A_l}(\sigma_{sc_1 \ldots sc_m}(R_1 \underset{jc_1}{\bowtie} R_2 \ldots R_n \underset{jc_n}{\bowtie} R_{n+1})),$$

where A_1, ..., A_l are attributes appearing in R_1, ..., and R_{n+1}, sc_i (i = 1, ..., m) is of the form: $B\ \alpha\ v$, or $B\ \alpha\ C$ (called the selection condition), and jc_k (k = 1, ..., n) is of the form: $B\ \alpha\ C$ (called the join condition), with B and C representing the attributes in R_1, ..., and R_{n+1}, v representing a constant and α being one of the operators $\{=, <, \leq, >, \geq, \neq\}$. The SQL's way of expressing such an algebra expression is

select A_1, ..., A_l
from R_1, ..., R_{n+1}
where sc_1 and ... sc_m and jc_1 and ... jc_n.

Then, the query decomposition can be done in an iteration process, in which each element (a relation name, an attribute name or a constant) appearing in the query is checked against the data dictionary.

First of all, we notice that in view of our pairwise integration process (see 2.2.4), we need only to consider the case that a global query is decomposed into two ones (which is called a binary decomposition hereafter) and there is no mixing appearances of local relations in a decomposed query. (But for a close cooperation, the mixing appearance of local relations should be handled; which is not reported here for ease of explanation.). Then, along an integration binary tree like that shown in Fig. 4(b), a recursive process of binary decompositions can be invoked to make a complete decomposition for the integration involving more than two component databases. A second point we should pay attention to is that for a binary decomposition at most two decomposed subqueries can be generated.

We have the following definition.

Definition An *intermediate query* is a (global) query changed so that at least one relation name in it is replaced with a local one.

Accordingly, a binary decomposition is a process to generate (two) intermediate queries, following a series of substitution operations to replace each element with its local counterparts. For a relation name appearing in a global query, we distinguish among four cases:

(1) there is an equivalence assertion is associated with it,
(2) there is an intersection assertion is associated with it,
(3) there is a derivation assertion is associated with it and
(4) there is no assertion is associated with it at all.

In terms of different cases, four decomposition strategies are developed.

Formally, it can be described as follows.

Algorithm *binary_decomposition*(q) (*q is a select-project-join query.*)
begin
generate_intermediate_queries(q); (*see below*)
let q_1 and q_2 be two intermediate queries generated by *generate_terminal_queries*(q);
for each q_i (i = 1, 2) **do**
substitution(q_i); (*see below*)
end

In the following, we give the algorithms for both *generate_intermediate_queries*(q) and *substitution*(q_i).

Algorithm *generate_intermediate_queries*(q)
begin
label := False; (**label* is used to control the **while-do** loop.*)
r := the first relation name appearing in q;
while *label* = False **do**
{**if** there exists an equivalence assertion associated with r in the assertion set **then**
{let the assertion be of the form: $r_1 \equiv r_2$;
generate two queries q_{r1} and q_{r2} by replacing all the r's appearances in q with
r_1 and r_2, respectively;
label = True;}
if there exists an intersection assertion associated with r in the assertion set **then**
{let the assertion be of the form: $r_1 \cap r_2$;
generate two queries q_{r1} and q_{r2} by replacing all the r's appearances in q with
r_1 and r_2, respectively;
let *new_element* be the new element constructed for $r_1 \cap r_2$; (*note that *new_element* can
be found in class "*new elements*" in the data dictionary.*)
for each select or join condition *Con* in q **do**
{let a be an attribute involved in *Con*;
if a appears in *new_element* and is involved in some *"method"*defining a "*global*" attribute **then**
remove *Con* from q_{r1} and q_{r2}, respectively; (*The reason for this is given below.*)
insert a into q_{r1} and q_{r2} as project attributes, respectively;}
label = True;}
if there exists a derivation assertion associated with r in the assertion set **then**
{let the assertion be of the form: $r_1, ..., r_n \rightarrow r$;
generate q_{r1} by replacing r's appearances in q with $r_1, ..., r_n$
(also the corresponding join conditions among r_is should be added to q_{r1}); (*see Example 5*)
generate q_{r2} by replacing r's appearances in q with r; (*That is, q_{r2} is simply a copy of q.*)
label = True;}
if there is no equivalence, intersection or derivation assertion associated with r **then**
r := next(r);} (*in this case, the next relation name will be checked.*)
end

Note that in the above algorithm, $\varnothing$ and $\subseteq$ are not considered for the decomposition of a select-project-join query, since if two concepts are associated with $\varnothing$ or $\subseteq$, only one of them is involved in the query each time. (But they should be considered for the new integrity constraints.) Further, for the intersection assertion, a select or a join condition will be removed from q_{r1} and q_{r2} if at least one attribute appearing in it is involved in the definition of some new (also global) attribute for the integrated schema (see 2.2.4). The reason for this is that the check of the condition can not be made until the corresponding local attribute values are available and computed in terms of the definition of the new attribute. Thus, such removed conditions are neglected only for the time being and should be considered once again during the synthesis process (see 3.3). Accordingly, the corresponding attributes are shifted to the project-range (as the project attributes) in the query.

Example 3. Consider a global query: $q = \pi_{\text{name, income}}(\sigma_{\text{income>1000}\wedge\text{research\_area='informatik'}}(\textit{faculty}))$, where *faculty* is a global relation with three attributes: "name", "income" and "research_area". If an assertion of the form: $S_1\bullet\textit{faculty} \cap S_2\bullet\textit{student}$ is declared and the corresponding new element is constructed in the data dictionary as given in Example 1, two intermediate queries: $q_1 = \pi_{\text{name, income}}(\sigma_{\text{research\_area='informatik'}}(\textit{faculty}))$ and $q_2 = \pi_{\text{name, income}}(\sigma_{\text{research\_area='informatik'}}(\textit{student}))$ will be generated by *generate_intermediate_queries*(q). Note that for them select condition "income>1000" is eliminated and "income" is accordingly moved to the project-range to get the relevant local values.

Example 4. Consider the global query given in Example 3 again. If we have the new elements stored as in Example 2, i.e., a new attribute of the form: {*work_area*, $S_1\bullet\textit{faculty}\bullet\textit{research\_area}$, $S_2\bullet\textit{student}\bullet\textit{study\_area}$, _, *m*''} is also defined in "*new elements*", then two intermediate queries: $q_3 = \pi_{\text{name, income, work\_area}}(\textit{faculty})$ and $q_4 = \pi_{\text{name, income, work\_area}}(\textit{student})$ will be produced by the above algorithm.

Example 5. Assume that we have an assertion of the form: $S_1(\textit{parent}, \textit{brother}) \rightarrow S_2\bullet\textit{uncle}$ stored in the data dictionary. Consider a global query: $q = \pi_{\text{name}}(\sigma_{\text{nephew='John'}}(\textit{uncle}))$. If the relation schemas of *parent* and *brother* are *parent*(name, children) and *brother*(bname, brothers), then q_1 (the query against S_1) will be of the form: $\pi_{\text{bname}}(\sigma_{\text{children='John'}}(\textit{parent} \underset{\text{name=brothers}}{\bowtie} \textit{brother}))$, while q_2 (the query against S_2) is the same as q. We notice that to generate a query like q_1 automatically, we have to make the join conditions among the relations appearing in the left-hand side of "$\rightarrow$" and the correspondences of the attributes of the both sides available beforehand. Therefore, they should be stored along with the corresponding derivation assertions in the data dictionary.

After the first decomposition step, two intermediate queries are produced and a substitution process will be executed to replace each "integrated" element in them with the corresponding local one. Obviously, this can be done in a similar way to that in *generate_intermediate_queries*(q). (In the following algorithm, q_i represents the query issued to DB*i*.)

Algorithm *substitution*(q_i) (*All the global elements in q_i will be replaced so that it can be evaluated in DB_i.*)
begin
for each element (relation or attribute name) e in q_i **do**
{**if** there exists an equivalence assertion associated with e in the assertion set **then**
{let the assertion be of the form: $e_1 \equiv e_2$;
replace e in q_i with e_i;}

if there exists an intersection assertion associated with e in the assertion set **then**
{let the assertion be of the form: $e_1 \cap e_2$;
replace e in q_i with e_i;
if e is a (global) relation name **then**
{let *new_element* be the new element constructed for $e_1 \cap e_2$; (*note that *new_element* can be found in class "*new elements*" in the data dictionary.*)
for each select or join condition *Con* in q **do**
{let a be an attribute involved in *Con*;
if a appears in *new_element* and is involved in some method **then**
remove *Con* from q_i;
insert a into q_i;}}}
if there exists a derivation assertion associated with e in the assertion set **then**
{let the assertion be of the form: $e_1, ..., e_n \rightarrow e$;
replace e's appearances in q_i with $e_1, ..., e_n$ if $e_1, ..., e_n$ are relation names in DB_i;
(also the corresponding join conditions among e_is should be added to q_i);
if there is no equivalence, intersection or derivation assertion associated with e **then**
{if DB_i is taken as the basic integrated version, nothing will be done;
else remove e and those select and join conditions involving any attribute of e;}}
end

Example 6. Consider the intermediate queries q_1 and q_2 of Example 3. If an assertion of the form: research_area ≡ study_area exists (but no new element for it) in the data dictionary, then q_1 and q_2 will be changed into the forms: $\pi_{\text{name, income}}(\sigma_{\text{research\_area='informatik'}}(\textit{faculty}))$ and $\pi_{\text{name, income}}(\sigma_{\text{study\_area='informatik'}}(\textit{student}))$, respectively. If the intermediate queries are q_3 and q_4 of Example 4, they will be changed into the forms: $\pi_{\text{name, income, research\_area}}(\textit{faculty})$ and $\pi_{\text{name, income, study\_area}}(\textit{student})$, respectively.

3.2 Query translation

If the relevant RSTs and derivation rules are stored in the data dictionary, a query submitted to an integrated schema can be translated automatically. In the following, we first introduce an important concept, the so-called *extended substitution* in 3.2.1. Then, in 3.2.2, we demonstrate our strategy for the translation of queries involving no joins. In 3.2.3, the translation of joins is simply discussed.

3.2.1 Extended substitutions

Note that an attribute involved in such an algebra expression may either appear in sc_i (i = 1, ..., m), or/and in $\{A_1, ..., A_l\}$, or not be involved in any operation at all. To characterize this feature, we associate each attribute with a label which consists of a subset $ap \subseteq \{p, s, ni, V\}$, where p, s, ni and V stand for 'project', 'select', 'not-involved' and 'the current values of the attribute', respectively.

In terms of an algebra expression q, we can instantiate the variables appearing in the consequent part of a derivation rule which matches q. Then, by the constant propagation, the antecedent part of the rule will also be instantiated; and what we want now is to derive a set of new algebra expressions in terms of it.

Unfortunately, such a derivation can not be done only by the constant propagation, since both the higher order information (e.g., about iterations over relation/attribute names) and the necessary control mechanism are absent. For this purpose, we introduce the concept of extended substitutions.

Definition (*assumed values*) A assumed value (for some variable) is either of the form: $\underline{\alpha X}$ or

$\in S$, where X is either a variable or a constant, $\alpha \in \{=, <, \leq, >, \geq, \neq\}$ and S represents a set of constants.

For example, $>x$, $=c$ and $\in \{c_1, c_2, ..., c_n\}$ are three assumed values.

Definition (*extended substitutions*) An extended substitution (ES) is a finite set of the form: $\{x_1/v_1, ..., x_l/v_l\}$, where x_i $(i = 1, ..., l)$ is a variable and v_i $(i = 1, ..., l)$ is a set of pairs of the form: $(Prop, v)$, where $Prop \in \{p, s, ni, V, \_\}$ (here, "_" means "do not care") and v is an assumed value as defined above or "_". In contrast to the traditional substitution concept, the variables $x_1, ..., x_l$ may not be distinct, Each element x_i/v_i is called a binding for x_i and a variable may have several bindings.

For example, $\delta = \{x/\{(p, \_)\}, y/\{(p, \_), (\_, \in \{name_1', name_2', ..., name_m'\})\}, z/\{(s, >1000)\}$ is a legal ES. Alternatively, this ES can also be written as $\{x/(p, \_), y/(p, \_), y/(\_, \in \{name_1', name_2', ..., name_m'\}), z/(s, >1000)\}$.

3.2.2 Translation of simple algebra expressions: $\pi(\sigma(R))$

The translation can be pictorially illustrated as shown in Fig. 6.

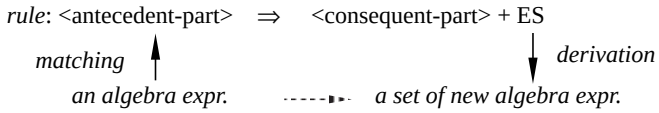

Fig. 6. Illustration for query translation process

In the following, we discuss this process in detail.

Essentially, this process consists of two functions. With the first function, we generate an ES by matching the algebra expression to be translated with the corresponding rule's antecedent part. With the second function, we derive a set of new algebra expressions in terms of the ES and the rule's consequent part. These two functions can be defined as follows:

the first function: *substi-production*: $\boldsymbol{P} \times \boldsymbol{A} \rightarrow \boldsymbol{S}$,
the second function: *expression-production*: $\boldsymbol{P} \times \boldsymbol{S} \rightarrow \boldsymbol{A}$,

where $\boldsymbol{P}$, $\boldsymbol{A}$ and $\boldsymbol{S}$ represent the set of all c-expressions, the set of all algebra expressions and the set of all extended substitutions, respectively.

Obviously, the matching algorithm used in Prolog [Ll87] can not be employed for our purpose and a bit modification is required so that not only the assumed values of a variable but more information associated with it are also evaluated. As we will see in the following algorithm (for *substi-production*), such information can be obtained by doing a simple analysis of the algebra expression to be translated (see lines 2-5). In the algorithm, the following definitions are used:

> *assumedValue*($A\alpha B$, T) returns an assumed value of the form: αX, where $A\alpha B$ is either a select condition or a join condition, T is an RST or a c-expression and $\alpha \in \{=, <, \leq, >, \geq, \neq\}$. X is a constant "c" if B = c, or a variable x if B:x is an attribute descriptor in T.
>
> $aV(q(x_1, ..., x_k), x_{i_j})$ returns an assumed value (for x_{i_j}), where q is a first order predicate, x_i $(i = 1, ..., k)$ may be a variables, a constant or a set of constants but $x_{i_j} \in \{x_1, ..., x_k\}$ must be a variable. For example, $aV(x \in \{c_1, c_2, ..., c_n\}, x) = \in \{c_1, c_2, ..., c_n\}$.

Algorithm *substi-production*(P, e) (*P is a c-expression and e is an algebra expression.*)
input: P: a c-expression; e: an algebra expression;

output: ES: an extended substitution;
begin
1 ES := ϕ;
2 **if** *e* is of the form: $\pi_{A_1 \ldots A_l}(\sigma_{sc_1 \ldots sc_m}(R))$ **then** {
3 construct three sets for *e*:
4 PA := {A_1, ..., A_l}; (*PA contains the attributes involved in project operations.*)
5 SC := {sc_1, ..., sc_i, ..., sc_m}; (*SC contains all select conditions.*)
6 **for** each $a \in$ PA **do**
7 {let *A*:*x* be an attribute descriptor of some RST in *P*;
8 **if** *a* = *A* **then** ES := ES ∪ {*x*/(*p*, _)}}
9 **for** each $a \in$ SC **do**
10 {let *A*:*x* be an attribute descriptor of some RST in *P*;
11 assume that *a* is of the form: *B* β *C*;
12 **if** *B* = *A* **then** {*v* := *assumedValue*(*B* β *C*, *P*); ES := ES ∪ {*x*/(*s*, *v*)};}}
13 **for** each predicate of the form: $q(x_1, ..., x_k)$ **do**
14 {**for** each variable x_{i_j} in *q* **do**
15 {*v* := *aV*(*q*(x_1, ..., x_k), x_{i_j}); ES := ES ∪ {x_{i_j}/(_, *v*)}}}
16 **if** *e* is of the form: *V*(*A*) **then** (**V*(*A*) is a query to inquire the current attribute values of *A*.*)
17 {let *A*:*x* be an attribute descriptor of some RST in *P*;
18 ES := ES ∪ {*x*/(*V*, _)};}
end

Example. Consider the algebra expression $e = \pi_{\text{income}}(\sigma_{\text{name='John'} \wedge \text{research\_area='Informatik'}}(R_1))$. If we want to translate it into an algebra expression which can be evaluated against DB_2 shown in Example 1, the rules for specifying the semantic discrepancies between DB_1 and DB_2 will be considered and the matching rule is $r_{DB1\text{-}DB2}$. Its antecedent part *P* is of the form: ["faculty"| name: *x*, research_area: *y*, income: *z*], $x \in \{name_1, name_2, ..., name_n\}$.

First, by executing lines 2-5, we will have

PA = {income},
SC = {name = 'John', research_area = 'Informatik'}.

Then, by executing lines 6-8, we will have

ES = {*z*/(*p*, _)}.

Next, after lines 9-12 are performed, ES will be of the following form:

ES = {*z*/(*p*, _), *y*/(*s*, ='Informatik'), *x*/(*s*, ='John')}

Finally, by executing lines 13-15, a new item: $x/\{(\_, \in \{name_1, name_2, ..., name_n\}$ (constructed in terms of the predicate: $x \in \{name_1, name_2, ..., name_n\}$) will be inserted into ES. Therefore, the final ES is of the form:

$\{z/(p, \_), y/(s, =\text{'Infomatil'}), x/\{(s, =\text{'John'}), (\_, \in \{name_1, name_2, ..., name_n\})\}\}$.

Note that in the final ES, pair $(\_, \in \{name_1, name_2, ..., name_n\}$ should be eliminated if 'John' $\in \{name_1, name_2, ..., name_n\}$ holds, since *x* = 'John' subsumes "$x \in \{name_1, name_2, ..., name_n\}$". In addition, if 'John' does not belong to $\{name_1, name_2, ..., name_n\}$, *substi-production* should report an "nil" to indicate that the matching does not succeed and the translation can not be made in terms of the rule. In fact, if $\{name_1, name_2, ..., name_n\}$ does not contain 'John', any query concerning 'John' submitted to DB_2 will evaluate to "nil". In the algorithm, however, such checks are not described for simplicity. It is easy to extend this algorithm to a complete version.

After the ES is evaluated, we can derive a set of new algebra expressions in terms of it and the

RSTs and the first-order predicates appearing in the consequent part of the rule. This can be done by executing the following algorithm, which generates not only two sets PA and SC (from them, an algebra expression can be constructed), but also a set iteration control statements with the form: for ... do, a set of checking statements with the form: if ... then, and a set of print statements. Together with PA and SC produced by the algorithm, such statements make us able to generate a complete query.

The main idea of it is as follows.

Consider a variable appearing in an RST. It may be a variable ranging over the relation names, a variable ranging over the attribute names or a variable ranging over some attribute values. Then, in terms of its bindings recorded in the corresponding ES, we can immediately fix its assumed value. On the other hand, which statements are associated with it can also be determined by a synthetic analysis of its assumed value and its properties.

Algorithm *expression-production*(*P*, δ) (**P* is a c-expression and δ is an ES.*)
input: *P*: a c-expression; δ: an ES;
output: PA: project attributes; SC: select conditions; FS: iteration control statements;
CS: checking statements;
begin
1 SC := ϕ; PA := ϕ; FS := ϕ; CS := ϕ; (*SC, PA, FS and CS are global set variables.*)
2 construct V_1, a set of variables (in *P*) ranging over attribute values;
3 construct V_2, a set of variables (in *P*) ranging over attribute names;
4 construct V_3, a set of variables (in *P*) ranging over relation names;
5 **for** each $x \in V_1$ **do**
6 call *attr-value-handling*(*x*, *P*, δ);
7 **for** each $x \in V_2$ **do**
8 call *attr-or-rel-name-handlin*(*x*, *P*, δ, 0);
9 **for** each $x \in V_3$ **do**
10 call *attr-or-rel-name-handlin*(*x*, *P*, δ, 1);
end

From the above algorithm, we see that two subprocedures will be called to deal with different cases. That is, *attr-value-handling* is used to tackle the variables ranging over attribute values and *attr-or-rel-name-handlin* is employed to deal with the variables ranging over attribute names or the variables ranging over relation names. Below we give a formal description for each. First, we define the following operation:

conditionProduction(*x*α*y*) returns a select condition or a join condition of the form: *E*α*F* if *E*:*x* and *F*:*y* are two attribute descriptors in the corresponding c-expression.

Algorithm *attr-value-handling*(*x*, *P*, δ)
begin
1 let *A*:*x* be an attribute descriptor of some RST in *P*; (*Here *A* is an attribute name or a variable.*)
3 **if** *x*/(*p*, _) is a binding in δ **then** PA := PA ∪ {*A*};
4 **if** there exist bindings: *x*/(*s*, v_1), ..., *x*/(*s*, v_k) in δ **then**
5 {**for** *i* = 1, ..., *k* **do**
6 {sc_i := *conditionProduction*(xv_i); SC := SC ∪ {sc_i};}
7 **if** *x*/(*V*, _) is a binding in δ **then** returns the attribute values of *A*;
end

Algorithm *attr-or-rel-name-handling*(*x, P,* δ, Int)
begin
0 **If** Int = 0 **then** find *x*:*z*, which is an attribute descriptor of some RST in *P*
1 **else** find [*x* | ...], which is an RST in *P*;
2 **if** there exist *x*/(*s*, v_1), ..., *x*/(*s*, v_k) in δ **then**
3 {**for** *i* = 1, ..., *k* **do**
4 {**if** v_i is of the form: =c **then** replace *x* with c in all the newly produced data structures
5 **else**
6 {let v_i is of the form: α*X*
7 generate a statement of the form: if *x*α*X* then;}}} (*produce a checking statement*)
8 **if** there exist a binding of the form: *x*/(_, ∈ {c_1, c_2, ..., c_m}) **then**
9 generate a statement of the form: for each *x* ∈ {c_1, c_2, ..., c_m} do; (*produce an iteration statement*)
10 **if** there exist bindings *x*/(_, v_1'), ..., *x*/(_, v_l') in δ with each v_i' ≠ "∈ {c_1, c_2, ..., c_m}" **then**
11 {**for** *i* = 1, ..., *l* **do** {generate a statement of the form: if *x*v_i' then;}}
12 **if** there exist a binding of the form: *x*/(*p*, _) in δ **then**
13 generte a output statement of the form: print(*x*);
14 **if** there exist a binding of the form: *x*/(*V*, _) in δ **then**
15 **if** Int = 0 **then** returns all the attribute names, over which *x* ranges;
16 **else** returns all the relation names, over which *x* ranges;
end

The result of these algorithms can be thought of as composed of four parts: a set of iteration control statements, a set of checking statements, a set of printing statements and an algebra expression derived from PA, SC and JC produced by the algorithm. If for each variable *x* (in the algebra expression) ranging over the relation names or ranging over the attribute names, there is a statement of the form: for each *x* ∈ {c_1, c_2, ..., c_m} do, where c_1, c_2, ..., c_m are constants, this result corresponds to a program which can be correctly executed. We do this as follows.

First, we suffix each iteration statement and each checking statement with an open bracket "{" and suffix each printing statement with a semi-comma. Then, change the newly generated algebra expression *e*' with "if *e*' then" and suffix it with "{". Next, we put them together in the order: iteration statements - checking statements - algebra expression - printing statements. Finally, we put the same number of close brackets "}" at the end of the sequence of the elements. For example, for the algebra expression $e = \pi_{\text{name, research\_area}}(\sigma_{\text{income}> 1000}(\text{"faculty"}))$, the following elements will be generated in terms of rule $r_{\text{DB1-DB3}}$:

"for each *y* ∈ {$name_1$', $name_2$', ..., $name_m$'} do",
"print(y)",
"$\pi_{\text{research\_area}}(\sigma_{\text{income}>1000}(y)\}$".

Then, the corresponding code will be of the form:

for each *y* ∈ {$name_1$', $name_2$', ..., $name_m$'} do
{if $\pi_{\text{research\_area}}(\sigma_{\text{income}>1000}(y)$ then
{print(y);}}.

According to the above discussion, the entire process for translating a simple algebra expression of the form: π(σ(R))) can be outlined as follows.

Algorithm *simple-query-translation*(*r, e*)
input: *r*: a derivation rule; *e*: an algebra expression;
output: a program corresponding to the translated query;
begin

δ := *substi-production*(antecedent-part of *r*, *e*);
S := *expression-production*(consequent-part of *r*, δ);
generate a program in terms of *S*;
end

3.2.3 About the translation of queries containing joins

Based on the technique proposed in 3.2.2, a simple but efficient method for translating queries involving joins can be developed as follows. Consider the algebraic expression $\pi_{A_1 \ldots A_l}(\sigma_{sc_1 \ldots sc_m}(R_1 \underset{jc_1}{\bowtie} R_2 \ldots R_n \underset{jc_n}{\bowtie} R_{n+1}))$ again. It can be rewritten into a set of expressions of the following form:

$T_1 = \pi_{A_{i_1} \ldots A_{i_r}}(\sigma_{sc_{j_1} \ldots sc_{j_s}}(R_1))$,
$T_2 = \pi_{A_{k_1} \ldots A_{k_t}}(\sigma_{sc_{l_1} \ldots sc_{l_u}}(R_2))$,
... ...
$T_{n+1} = \pi_{A_{m_1} \ldots A_{m_w}}(\sigma_{sc_{n_1} \ldots sc_{n_v}}(R_{n+1}))$,
$T = \pi_{A_1 \ldots A_l}(\sigma_{sc_1 \ldots sc_m}(T_1 \underset{jc_1}{\bowtie} T_2 \ldots T_n \underset{jc_n}{\bowtie} T_{n+1}))$,

where each T_i represents a subquery involving only one R_i, and therefore contains no joins.

Note that this rewriting is completely consistent with the traditional optimal technique [EN89] and can be implemented without difficulty. Then, we apply the technique discussed in 3.2.2 to each T_i and subsequently make a series of join operations between T_i's. In this way, each local result can be obtained correctly and efficiently.

3.3 Synthesis process

From the query decomposition discussed in 3.1, we see that a synthesis process is needed to get the final result of a query. The main reason for this is the existence of new elements for an integrated schema, which can not be computed until some local values are available. Therefore, during the query decomposition phase, the relevant select or join conditions are removed to avoid any incorrect checks. But now they should be considered. For example, global query $q = \pi_{\text{name, income}}(\sigma_{\text{income}>1000 \wedge \text{research\_area='informatik'}}(\textit{faculty}))$ may be decomposed into $q_1 = \pi_{\text{name, income}}(\sigma_{\text{research\_area='informatik'}}(\textit{faculty}))$ and $q_2 = \pi_{\text{name, income}}(\sigma_{\text{research\_area='informatik'}}(\textit{student}))$ during the decomposition phase. Further, q_1 may translated into

for each $y \in \{\text{name}_1, \text{name}_2, ..., \text{name}_n\}$ **do**
{**if** $\pi_y(\sigma_{y>1000}(\text{research})$ **then**
{print(*y*)}},

if the local database is like DB2. Assume that the returned results from the local databases are stored in s_1 and s_2. Then, s_1 is a set of pairs of the form: (*a*, *b*), where *a* represents a faculty member whose research area is informatik, and *b* is his income. Similarly, s_2 is also a set of pairs of the form: (*a*', *b*'), where *a*' and *b*' represent a student's name (whose study area is also informatik) and his financial support, respectively. In terms of the corresponding method defined on income, condition "income>1000" can be rewritten to *g*(*b*, *b*') > 1000 (see Example 1 for g's definition). Applying this condition to s_1 and s_2, we can get part results *s* to $\pi_{\text{name, income}}(\sigma_{\text{income}>1000 \wedge \text{research\_area='informatik'}}(\textit{faculty}))$, which belong to "faculty" and "students" simultaneously. The other part results belong to $IS_{\textit{faculty-}}$ ($= S_1 \bullet \textit{faculty} \cap \neg IS_{\textit{faculty, student}}$),

which can be obtained by applying the condition "income>1000" to s_1 - s. In addition, we notice that if no new element is involved in the query evaluation, the final results are the union of those from local databases.

In terms of the above analysis, we give our synthesis algorithm.

Algorithm *synthesis*
begin
let $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$ be two local results;
if no new element is involved during the query decomposition **then**
$\boldsymbol{s} := \boldsymbol{s}_1 \cup \boldsymbol{s}_2$;
else
for each $t_1 \in \boldsymbol{s}_1$ **do**
for each $t_2 \in \boldsymbol{s}_2$ **do**
for each $con(a_1, ..., a_n) \in Cons$ **do**
{let b_i and c_i (i = 1, ...n) be the two local counterparts of a_i;
let m_i be the method defined over b_i and c_i;
apply $con(m_1(b_1, c_1), ..., m_n(b_n, c_n))$ to t_1 and t_2;}
let $\boldsymbol{s}$' be result;
$\boldsymbol{s} := \boldsymbol{s}_1$ - $\boldsymbol{s}$' (or $\boldsymbol{s} := \boldsymbol{s}_2$ - $\boldsymbol{s}$', depending on which local database the relation name belongs to);
apply $con(b_1, ..., b_n)$ to $\boldsymbol{s}$ (or apply $con(c_1, ..., c_n)$ to $\boldsymbol{s}$ if $\boldsymbol{s} := \boldsymbol{s}_2$ - $\boldsymbol{s}$');
let $\boldsymbol{s}$'' be the result;
$\boldsymbol{s} := \boldsymbol{s}' \cup \boldsymbol{s}''$;
end

In the algorithm *Cons* represents set of all the select and join conditions removed during the query decomposition) and $con(a_1, ..., a_n)$ represents a select or a join condition involving attribute names a_1, ..., and a_n.

4. Conclusion

In this paper, a systematic method for evaluating queries submitted to a federated database is presented. The method consists of four phases: syntactic analysis, query decomposition, query translation and result synthesis. If the meta information are well defined, the entire process can be done automatically. Especially, in the case of structure conflicts, the query translation can be made based on the relation structure terms and the corresponding derivation rules. To this end, two new concepts: assumed values and extended substitution are developed, which make the propagation of the structure information possible. The query decomposition is based on the correspondence assertions. In addition, a new assertion: derivation assertion is introduced, which enables us to get a semantically more complete integrated schema, i.e., more complete answers to a query issued to an integrated database can be obtained.

Reference

BOT86 Y. Breitbart, P. Olson, and G. Thompsom, "Database integration in a distributed heterogeneous database system," in *Proc. 2nd IEEE Conf. Data Eng.*, 1986, pp. 301 - 310.

CB96 Y. Chen and W. Benn, "On the Query Optimization in Multidatabase," in *Proc. of the first Int. Symposium on Cooperative Database Systems for Advanced Application*, Kyoto, Japan, Dec. 1996, pp. 137 - 144.

CW93 S. Ceri and J. Widom, "Managing Semantic Heterogeneity with Production Rules and Persistent Queues", in *Proc. 19th Int. VLDB Conference*, Dublin, Ireland, 1993, pp. 108 -119.

DS96 W. Du and M. Shan, "Query Processing in Pegasus," in: O. Bukhres, A.K. Elmagarmid (eds): *Object-oriented Multidatabase Systems: A Solution for Advanced Applications*. Chapter 14. Prentice Hall, Englewood Cliffs, N.J., 1996.

DSD94 W. Du, M. Shan and U. Dayal, “Reducing Multidatabase Query Response Time by Tree Balancing”, *DTD Technical Report*, Hewlett-Packard Labs., 1994.

DSD95 W. Du, M. Shan and U. Dayal, “Reducing Multidatabase Query Response Time by Tree Balancing”, in *Proc. 15th Int. ACM SIGMOD Conference on Management of Data*, San Jose, california, 1995, pp. 293 -303.

Du94 Y. Dupont, “Resolving Fragmentation conflicts schema integration,” in *Proc. 13th Int. Conf. on the Entity-Relationship Approach*, Manchester, United Kingdom, Dec. 1994, pp. 513 - 532.

EN89 R. Elmasri and S.B. Navathe, *Foundamantals of Database Systems*, The Benjamin/Cummings Publishing Company Inc. New York, 1989.

ETB96 C.J. Egyhazy, K.P. Triantis and B. Bhasker, “A Query Processing Algorithm for a System of Heterogeneous Distributed Databases”, *Int. Journal of Distributed and Parallel Databases*, 4, 49 - 79, Dec. 1996.

HLM94 G. Harhalakis, C.P. Lin, L. Mark and P.R. Muro-Medrano, “Implementation of Rule-based Information Systems for Integrated Manufacturing”, *IEEE Trans. on Knowledge and Data Engineering*, vol. 6, No. 6, 892 - 908, Dec. 1994.

KFMRN96 W. Klas, P. Fankhauser, P. Muth, T. Rakow and E.J. Neuhold, “Database Integration Using the Open Object-oriented Database System VODAK,” in: O. Bukhres, A.K. Elmagarmid (eds): *Object-oriented Multidatabase Systems: A Solution for Advanced Applications*. Chapter 14. Prentice Hall, Englewood Cliffs, N.J., 1996.

Jo93 P. Johannesson, “Using Conceptual Graph Theory to Support Schema Integration”, in *Proc. 12th Int. Conf. on the Entity-Relationship Approach*, Arlington, Texas, USA, Dec. 1993, pp. 283 - 296.

LA86 W. Litwin and A. Abdellatif, “Multidatabase interoperability,” *IEEE Comput. mag*., vol. 19, No. 12, pp. 10 - 18, 1986.

LHSC95 E. Lim, S. Hwang, J. Srivastava, D. Clements and M. Ganesh, “Myriad: design and implementation of a federated database prototype”, *Software-Practice and Experience*, Vol. 25(5), 533 - 562, May 1995.

LOG93 H. Lu, B. Ooi and C. Goh, “Multidatabase Query Optimization: Issues and Solutions”, *In Proc. of 3th Int. Workshop on Research Issues in Data Engineering*, pp. 137 - 143, Vienna, Austria, April 1993.

LHS95 E. Lim, J. Srivastava and S. Hwang, “An Algebraic Transformation Framework for Multidatabase Queries,” *Distributed and Parallel Databases*, Vol. 3, 273 - 307, 1995.

Ll87 J.W. Lloyd, “*Foundation of Logic Programming*”, Springer-Verlage, Berlin, 1987.

LNE89 J.A. Larson, S.B. Navathe, and R. Elmasri, “A theory of attribute equivalence in databases with application to schema integration,” *IEEE Trans. Software Eng*., vol. 15, No. 4, pp. 449 - 463, 1989.

LP96 L. Liu and C. Pu, “Issues on Query Processing in Distributed and Interoperable Information Systems,” in: *Proc. of the first Int. Symposium on Cooperative Database Systems for Advanced Application*, Kyoto, Japan, Dec. 1996, pp. 218 - 227.

LW96 C. LEE and M. Wu, “A Hyperrelational Approach to Integration and Manipulation of Date in Multidatabase Systems,” *Int. Journal of Cooperative Information Systems*, Vol. 5, No. 4 (1996) 395-429.

NTA96 I. Nishizawa, A. Takasu and J. Adachi, “A query Processing Method for Integrated Access

to Multiple Databases," in: *Proc. of the first Int. Symposium on Cooperative Database Systems for Advanced Application*, Kyoto, Japan, Dec. 1996, pp. 385 - 399.

Ra87 T. S. Ramkrishna, *UNIX utilities*, McGraw-Hill, New York, 1987.

RPRG94 M.P. Reddy, B.E. Prasad, P.G. Reddy, and A. Gupta, "A methodology for integration of heterogeneous databases," *IEEE Trans. on Knowledge and Data Engineering*, vol. 6, No. 6, 920 - 933, Dec. 1994.

SC94 P. Scheuermann and E.I. Chong, "Role-based query processing in multidatabase systems", in: *Proc. of 4th Int. Conf. on Extending Database Technology*, Cambridge, United Kingdom, March 1994, pp. 95 - 108.

SK92 W. Sull and R.L. Kashyap, "A self-organizing knowledge representation schema for extensible heterogeneous information environment," *IEEE Trans. on Knowledge and Data Engineering*, vol. 4, No. 2, 185 - 191, April 1992.

SPD92 S. Spaccapietra and P. Parent, and Yann Dupont, "Model independent assertions for integration of heterogeneous schemas", *VLDB Journal*, No. 1, pp. 81 - 126, 1992.

SP94 S. Spaccapietra and P. Parent, "View integration: a step forward in solving structural conflicts", *IEEE Trans. on Knowledge and Data Engineering*, vol. 6, No. 2, 258 - 274, April 1994.