Building DD to Support Query Processing in Federated Systems

Yangjun Chen and Wolfgang Benn
Computer Science Dept., Technical University of Chemnitz-Zwickau,
09107 Chemnitz, Germany
{chen, Benn} @informatik.tu-chemnitz.de

Abstract

In this paper, a method for building data dictionaries (DDs) is discussed, which can be used to support query processing in federated systems. In this method, the meda data stored in a DD are organized into three classes: structure mappings, concept mappings and data mappings. Based on them, a query submitted to a federated system can be decomposed and translated, and the local results can be synthesised automatically.

1 Introduction

Due to the rapid advance in networking technologies and the requirement of data sharing among different organizations, federated systems have become the trend of future database developments [BOT86, LA86, Jo93, SK92, CW93, HLM94, RPRG94, KFMRN96]. The research on this issue can be roughly divided into two main categories: the tightly-integrated approach that integrates databases by building an integrated schema and the loosely-integrated approach that achieves interoperability by using a multidatabase language. The method proposed here belongs to the second category, but providing the possibility to build integrated schemas. The key idea of this method is to construct a powerful data dictionary to govern the semantic conflicts among the local databases. We recognize three classes of meta data stored in a data dictionary: structure mappings, concept mappings and data mappings, each for a different kind of semantic conflicts: structure conflict, concept conflict and data conflict. Based on such meta informations, a query submitted to a federated system can be translated (in terms of structure mappings), decomposed (in terms of concept mappings) and synthesised automatically. In addition, for the execution optimization, some new techniques are developed for generating balanced join trees, which are quite different from those used in distributed databases and in parallel processing of joins.

The remainder of the paper is organized as follows. In Section 2, we show our system architecture to provide a background for the subsequent discussions. Then, in Section 3, we discuss the mata data classification and the data dictionary structure. In section 4, we present our strategies for query processing, in cluding query decomposition, query translation, query optimization and result synthsis. Section 5 is a short summary.

2 System Architectur

In this section, we show our system architecture and its installation.

2.1 System Logical Architectur

Our system architecture consists of three-layers: FSM-client, FSM and FSM-agents as shown in Fig. 1. 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, as well as the global query treatment. In addition, a centralized management is

Proceedings of the 4th KRDB Workshop
Athens, Greece, 30-August-1997

(F. Baader, M.A. Jeusfeld, W. Nutt, eds.)

http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-8/

supported at this layer. The FSM-agents layer corresponds to the local system management, addressing all the issues w.r.t. schema translations as well as local transaction and query processing. (Here FSM stands for "Federated System Manager".)

According to 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 a database named "PatientDB", installed in "FSM-agent1".

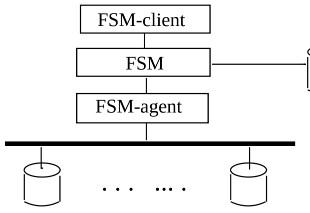

Figure 1: System Architecture

For ease of exposition, in the following, we discuss the query optimization in a simple setting that each local database involved in a query is relational.

2.2 System Installation

Fig. 2 shows an experiment environment, in which our system is installed.

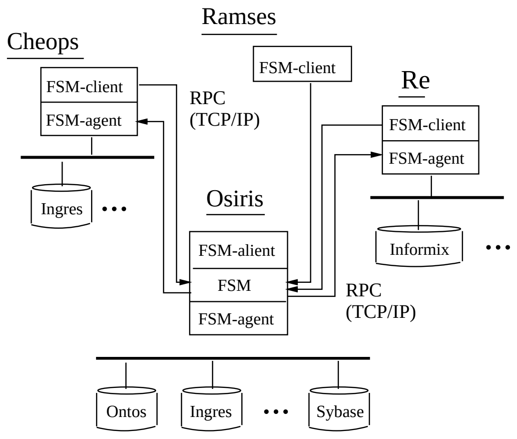

Figure 2: System Installation

Here Cheops, Ramses, Re and Osiris are the names of four computers located at different sites and in each computer several databases are installed. Since in our system the data dictionary itself is implemented as an object oriented database, e.g., as an ONTOS database, the FSM layer can only be installed in those machines where ONTOS is available. In contrast, the FSM-agent layer should be installed in any machine if some of its databases participate in the integration. At last, we make the FSM-client layer available in each machine so that the system can be manipulated at any site. To this end, we have implemented our own communication protocol using RPC (remote procedure call [Bl92]) which works in a server-client manner.

3 Meta Data and Data Dictionary

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.

3.1 Structure Mappings

In the case of relational databases, we consider three kinds of structure conflicts which can be illustrated as shown in Fig. 3.

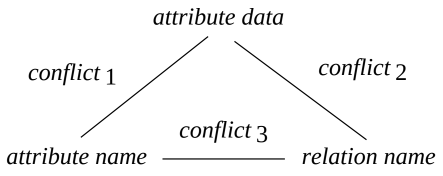

Figure 3: Illustration for Structure Conflicts

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, and
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 $\pi_{name,research\_area}(\sigma_{income>1000}(R))$ 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**
$\{\pi_{name,research\_area}(\sigma_{income>1000}(y))\}$.

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:

$$[re_{\{R_1,...,R_m\}}|a_1 : x_1, a_2 : x_2, ..., a_l : x_l, y : z_{\{A_1,...,A_n\}}],$$

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 relation. For example, $[re_{\{name'_1,...,name'_m\}}|$ research_area: x, income: y] represents any relation in DB$_3$, while an RST of the form: $[re_{\{research\}}$ | research_area: x, y: $z_{\{name_1,...,name_n\}}$] (or simply ["research" | research_area: x, y: $z_{\{name_1,...,name_n\}}$]) 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: $\gamma_1 \& \gamma_2$ ... $\& \gamma_l \Leftarrow \tau_1 \& \tau_2$ ... $\& \tau_k$, 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$_{DB_1-DB_3}$: [y| research_area: x, income: z] $\Leftarrow$
["faculty"|name: y, research_area: x, income: z],
$y \in$ {name$'_1$, name$'_2$, ..., name$'_m$},
r$_{DB_3-DB_1}$: ["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$_{DB_3-DB_1}$, 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$_{DB_1-DB_2}$: ["research" | research_area: y, x: z] $\Leftarrow$
["faculty"|name: x, research_area: y, income: z],
$x \in$ {name$_1$, name$_2$, ..., name$_n$},
r$_{DB_2-DB_1}$: ["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$_{DB_3-DB_2}$: ["research" | research_area: x, y: z] $\Leftarrow$
[y|research_area: x, income: z],

$y \in \{\text{name}_1, \text{name}_2, ..., \text{name}_n\}$,

$\text{r}_{DB_2-DB_3}$: $[y|$ research_area: x, income: $z] \Leftarrow$

["research"|research_area: x, y: z],

$y \in \{\text{name}'_1, \text{name}'_2, ..., \text{name}'_m\}$,

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: $A \Leftarrow B$, B and A are called the antecedent part and the consequent part of the rule, respectively.

3.2 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 ($\equiv$), inclusion ($\supseteq$ or $\subseteq$), disjunction (ϕ) and intersection ($\cap$). 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; and $S_1 \bullet A \phi S_2 \bullet B$, 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: $S_1 \bullet person \equiv S_2 \bullet human$, $S_1 \bullet book \subseteq S_2 \bullet publication$, $S_1 \bullet faculty \cap S_2 \bullet student$, $S_1 \bullet man \phi S_2 \bullet woman$.

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 ($\rightarrow$) 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.

3.3 Data Mapping

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: $S_1 \bullet$height_in_inches $\equiv S_2 \bullet$ 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. 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:

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

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.

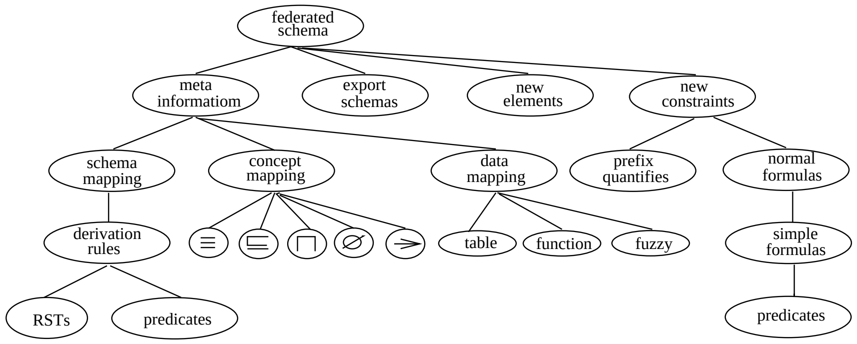

Figure 4: Data Dictionary

3.4 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. 4

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. 4). 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 $S_1 \bullet person \equiv S_2 \bullet human$ 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 $S_1 \bullet person$ and $S_2 \bullet human$ need to be considered.) However, if $S_1 \bullet faculty \cap S_2 \bullet student$ 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 $IS_{faculty,student} = S_1 \bullet faculty \cap S_2 \bullet student$, $IS_{faculty-} = S_1 \bullet faculty \cap \neg IS_{faculty,student}$ and $IS_{student-} = S_2 \bullet student \cap \neg IS_{faculty,student}$. 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: $\forall e$(in $Employee$)$\exists d$(in $Department$) ($d.name = e.Dept \rightarrow e.name$ 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 " $\Leftarrow$ ". 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 $C \in \mathbf{C}$, denoted by $type(C)$, is defined as:

$$type(C) = < a_1 : type_1, ..., a_l : type_l, Agg_1 \text{ with } cc_1 : out-typ_1, \\ ..., Agg_k \text{ with } cc_k : out-type_k, m_1, ..., m_h >$$

where a_i represents an attribute name, Agg_j represents an aggregation function: $C \rightarrow C'$ ($C, C' \in \mathbf{C}$ and $out{-}type_j \in type(\mathbf{C})$), 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

$type(\text{"}federated\ schema\text{"}) = <IS :< string >, S_f :< string >,$
$S_s :< string >, indicator :< boolean >,$
Agg_1 with $[1:1] :< type(\text{"}meta\ information\text{"})>,$
Agg_2 with $[1:2] :< type(\text{"}export\ schemas\text{"})>,$
Agg_4 with $[1:1] :< type(\text{"}new\ constraints\text{"})>>,,$

where IS is an attribute 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. 5(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.

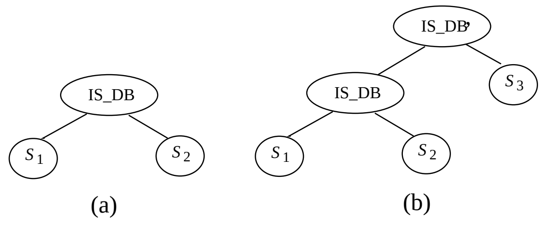

Figure 5: Integration Process

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. 5(b) can be recorded.

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

$type(\text{"}export\ schemas\text{"}) = <S :< string >, path :$
$<concatenation\ of\ strings>, r\_a\_names: <set\ of\ pairs>>,$

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:

$type(\text{"}meta\ information\text{"}) = <S_f\_S_s$:<pairs of strings>,
Agg_1 with $[1:n] :< type(\text{"}structure\ mapping\text{"})>,$
Agg_2 with $[1:n] :< type(\text{"}concept\ mapping\text{"})>,$
Agg_3 with $[1:n] :< type(\text{"}data\ mapping\text{"})>>,$

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 $IS_{faculty-} = S_1 \bullet faculty \cap \neg IS_{faculty,student}$.) Then, a set of functions has to be defined in *"new elements"*. In general, class *"new elements"* has the following structure:

$type(\text{"}new\ elements\text{"}) = <S :< string >,$
$new\_elem :< set >, m_1, ..., m_h >.$

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:

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

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:

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

and m' is another one for the function below:

$$f(x,y) = \begin{cases} \frac{x+y}{2}, & \text{if there exist tuple } t_1 \in faculty \\ & \text{tuple } t_2 \in student \text{ such that } t_1.name = \\ & t_2.name \text{ (in terms of data mapping),} \\ & \text{and } x = t_1.name \text{ and } y = t_2.name, \\ & \text{in terms of data mapping),} \\ null, & \text{otherwise.} \end{cases}$$

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

$$faculty(name, income, research\_area) \text{ and}$$
$$student(name, study\_support),$$

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:

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

Conversely, if the relation schemas of *faculty* and *student* are

$$faculty(name, income) \text{ and}$$
$$student(name, study\_support, study\_area),$$

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:

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

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:

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

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: $(p_{11} \vee ... p_{1n_1}) \wedge ... \wedge (p_{j1} \vee ... p_{jn_j})$), x_1, ..., x_n are all variables occurring in exp, and T_1, ..., T_n are set-valued expressions (or class names). Therefore, two classes *"prefix quantifier"* and *"normal formulas"* are defined as parts of *"new constraints"* (see Fig. 4). Then, class *"new constraints"* is of the following form:

$$type(\text{"new constraints"}) = < constraint\_number :< string >,$$
$$Agg_1 \text{ with [1:1]: } < type(\text{"prefix quantifier"})>,$$
$$Agg_2 \text{ with [1:2]: } < type(\text{"normal formulas"})>>,$$

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:

$$type(\text{"prefix quantifier"}) = < constraint_number :< string >,$$
$$quantifiers :< string >>,$$

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

$$type(\text{"normal formulas"}) = < constraint_number :< string >,$$
$$l\_formula \text{ with [1:n]: } < type(\text{"}formulas\text{"}) >,$$
$$r\_formula \text{ with [1:n]: } < type(\text{"}formulas\text{"}) >>,$$

where *quantifiers* is a single-valued attribute used to store a string of the form: $(Qx_1 \in T_1)...(Qx_n \in T_n)$, 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. 4 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 3.1 - 3.3 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 such meta informations stored in a data dictionary.

4 Query Processing

Based on the metadata built as above, a query submitted to an integrated schema can be evaluated in a five-phase method (see Fig. 6). First, the query

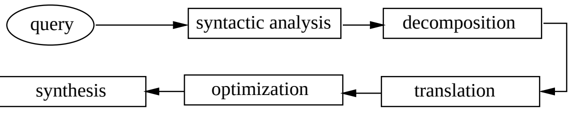

Figure 6: Query Processing

will be analyzed syntactically (using LEX unix utility [Ra87]). Then, it will be decomposed in terms of the correspondence assertions. Next, we translate any subquery in terms of the derivation rules so that it can be evaluated in the corresponding component database. In the fourth phase, we generate an optimal execution plan for each decomposed subquery. At last, a synthesis process is needed to combine the local results evaluated.

- *query decomposition*

After the syntactic analysis, a syntactically correct query will be decomposed in terms of the correspondence assertions, which can be pictorially illustrated as shown in Fig.7.

$$\begin{tikzcd}
\pi \ldots (\sigma \ldots (R_1 \bowtie \ldots \bowtie R_n)) \arrow[d, Rightarrow, "\text{query decomposition}"'] & \\
\begin{array}{l} \pi \ldots (\sigma \ldots (R_1^1 \bowtie \ldots \bowtie R_n^1)) \\ \ldots \ldots \\ \pi \ldots (\sigma \ldots (R_1^j \bowtie \ldots \bowtie R_n^j)) \\ \ldots \ldots \\ \pi \ldots (\sigma \ldots (R_1^{m_1} \bowtie \ldots \bowtie R_n^{m_n})) \end{array} & \boxed{\text{assertion set}} \arrow[ul, Rightarrow]
\end{tikzcd}$$

Figure 7: Query Decomposition

where each R_i stands for a global relation and each is a relation in some local database DB_j. Furthermore, we assume that $R_i = R_i^1 \cup R_i^2 \ldots \cup R_i^{m_i}$ for some m_i and an assertion of the form: $(\ldots(R_i^1 Q_{i1} R_i^2) \ldots\ Q_{im_i} R_i^{m_i})$ is declared, where each Q_{ij} represents an assertion $\equiv$, $\cap$ or $\subseteq$. Notice that such an assertion is built manually along with the integration process. In our system, the integration is always done pairwise as shown in Fig. 5(b). That is, for two local databases, say DB_1 and DB_2, we generate an integrated schema IS_1. Then, when a third local database DB_3 is about to be integrated, we glue it to IS_1 in the same way as for DB_2 to DB_1. Accordingly, corresponding to a global relation R_i', we may have an assertion $((R_i^1 Q_{i1} R_i^2) Q_{i2} R_i^3)$ established as shown in Fig. 8. By the query decomposition, there may be $m_1 \times m_2 \ldots \times m_n$ subqueries generated in total.

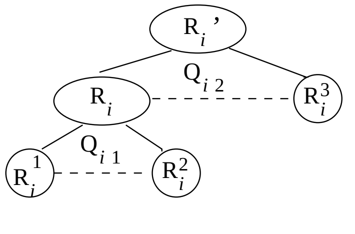

Figure 8: Relation Integration

- *query translation*

Each leaf node of the tree shown in Fig. 10(b) should be generally considered as a query of the form: $\pi(\sigma(R))$ since in terms of the traditional optimal strategy, the project and select operations should be shifted to be evaluated as early as possible. In addition, such a query should be evaluated in a local database. But in the presence of structure conflicts as demonstrated in 3.1, it has to be translated so that it can be evaluated locally. To this end, a mechanism is developed in our system to do the transformation automatically based on the relation structure terms and derivation rules discussed in 3.1. The mechanism can be illustrated as shown in Fig. 9.

$$\begin{tikzcd}
\textit{rule}: \langle\text{antecedent-part}\rangle \Rightarrow & \langle\text{consequent-part}\rangle + ES \arrow[d, "\textit{derivation}"] \\
\textit{an algebra expr.} \arrow[u, "\textit{matching}"] \arrow[r] & \textit{a set of new algebra expr.}
\end{tikzcd}$$

Figure 9: Illustration for Query Translation

where "ES" stands for "*extended substitution*", a data structure used to store the result of matching an algebra expression of the form: $\pi(\sigma(R))$ with the antecedent part of some rule (see [CB96a]). Then, in terms of this result and the consequent part of the corresponding rule, a set of new algebra expressions can be derived. In this way, the query is translated.

- *optimization*

As in a traditional database, for each subquery produced by the query decomposition, an execution plan should be generated and optimized. First, all the select and project operations should be arranged to be performed as early as possible, as for a normal query. Then, for the join operations, we use a two-phase method to optimize the execution process. In the first phase, we generate an optimal left-deep join tree for the corresponding join sequence (see Fig. 10(a)). This can be achieved using the approaches proposed in [CWY96] or in [YL89]. In the second phase, we translate the left-deep join tree into a balanced bushy join tree using the methods developed in [CB96b, CB97, DSD95] (see Fig. 10(b) for illustration.) But one may wonder why not to generate a balanced bushy join tree directly from a join sequence as done in [CYW96]. The reason for this is as follows:

(i) The bushy join tree generated (directly from a join sequence) by [CYW96] is not balanced. Therefore, an extra process is needed to balance such a tree just as for a left deep join tree.

(ii) The time complexity of the algorithm for finding such a bushy join tree (directly from a join sequence) is $O(n \cdot e)$, where n and e are the numbers of the relations and the corresponding joins involved in a subquery, while the time complexity of the algorithm for finding a left deep join tree is $O(n^2)$ (see [CYW96]). As we can see in [CB96b], a recursive algorithm can be implemented, which translates a left deep join tree into a balanced bushy join tree but requires only $O(n^2)$ time. Therefore, theoretically, the strategy developed based on the transformation of the left deep join trees will have a better time complexity.

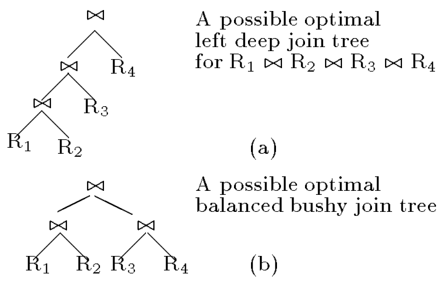
A possible optimal left deep join tree for $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$
(a)
A possible optimal balanced bushy join tree
(b)
R_1
R_2
R_3
R_4

Figure 10: Join Tree Transformation

- *synthesis process*

Normally, the synthesis process is very simple, by which all the local results are combined directly together. But in some cases, more complicated computations may be involved. For example, if *faculty* and *student* are two relations of two local relational databases DB_1 and DB_2, respectively, and an assertion of the form: $DB_1 \bullet faculty \cap DB_2 \bullet student$ is specified between them, then a query of the form:

$$\pi_{name,income}(\sigma_{income>1000 \wedge research\_area='informatik'}(Faculty))$$

($Faculty$ stands for the global version of *faculty* and *student*), submitted to the integrated version of DB_1 and DB_2, will be decomposed into two subqueries:

$$q_1 = \pi_{name,income}(\sigma_{income>1000 \wedge research\_area='informatik'}(faculty))$$
and
$$q_2 = \pi_{name,income}(\sigma_{income>1000 \wedge research\_area='informatik'}(student)).$$

However, if the attribute values of "income" is evaluated in terms of function $g(x, y)$ defined in 3.4, q_1 and q_2 have to be further changed into

$\pi_{name,salary}(\sigma_{research\_area='informatik'}(faculty))$ and $\pi_{name,study\_support}(\sigma_{research\_area='informatik'}(student))$.

We notice that by this modification, not only the global attribute name "income" is replaced with the corresponding local ones, but the condition "income > 1000" is also removed, since such a condition can not be checked until the corresponding local values are available. Obviously, the lost computation (due to the removing of "income > 1000" from the queries) has to be recovered in this phase.

5 Conclusion

In this paper, a systematic method for evaluating queries submitted to a federated database is outlined. The method consists of five steps: syntactic analysis, query decomposition, query translation, optimal execution plan generation, and synthesis process. If the metadata are well established, the entire process can be performed automatically.

References

[Bl92] J. Bloomer. *Power Programming with RPC* O'Reilly & Associates, Inc. 1992.

[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.

[CB96a] Y. Chen and W. Benn. "On the Query Translation in Federative Relational Databases," in: *Proc. of 7th Int. DEXA Conf. and Workshop on Database and Expert Systems Application,* Zurich, Switzerland: IEEE, Sept. 1996, pp. 491-498.

[CB96b] 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.

[CB97] Y. Chen and W. Benn. "Tree Balance and Node Allocation," accepted by *Int. Database Engineering and Application Symposium,* Montreal, Canada, Aug. 1997.

[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.

[CYW96] M-S. Chen, P.S. Yu and K-L. Wu. "Optimization of Parallel Execution for Multi-Join Queries," *IEEE Trans. on Knowledge and Data Engineering,* vol. 8, No. 3, June 1996, pp. 416-428.

[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.

[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.

[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.

[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.

[YL89] H. Yoo and S. Lafortune. "An Intelligent Search Method for Query Optimization by Semijoins," *IEEE Trans. on Knowledge and Data Engineering,* vol. 1, No. 2, June 1989, pp. 226 - 237.