An overview of the multidatabase manipulation language MDSL

W. Litwin, A. Abdellatif

INRIA

CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 535

AN OVERVIEW OF THE MULTIDATABASE MANIPULATION LANGUAGE MDSL

Witold LITWIN
Abdelaziz ABDELLATIF

Juin 1986

Résumé:

Avec l'application de plus en plus étendue du model relationnel, les données dont a besoin un usager font partie souvent de plusieurs bases de données sur differents sites et reseaux. Les propriétés logiques de telles données diffèrent de celles localisées dans une seule base. Ceci est une conséquence de la nouvelle forme de liberté dont disposent les possesseurs d'information pour satisfaire leurs besoins. Des cas pratiques montrent que les langages de manipulation de données doivent fournir des nouvelles fonctions. MDSL est un nouveau langage de manipulation de données proposant de telles fonctions. La plupart des fonctions de MDSL ne sont pas encore connues dans d'autres langages.

AN OVERVIEW OF THE MULTIDATABASE MANIPULATION LANGUAGE **MDSL**

Witold Litwin, Abdelaziz Abdellatif
INRIA, B.P 105, 78153 Le-Chesnay, France

ABSTRACT

With the relational model becoming widely applied, a user's data are more and more frequently in multiple relational databases on various computers and networks. The logical properties of such data differ from the classical ones within a single database. This is due to the new freedom of the data owner to privilege his own needs. Case studies show that data manipulation languages should then provide new functions. MDSL is a new data manipulation language proposing such functions. Most of MDSL functions are not yet known in other languages.

1. INTRODUCTION

The development of database systems (DBSs) has given rise to many databases. Frequently, dozens of databases exist on a large computer and thousands of databases are accessible through computer networks. In particular, videotex systems, like Prestel, Telelel or Telidon etc. provide hundreds of databases on almost any subject (Cinemas, restaurants, banking, trains, airlines, law).

While the purpose of the concept of a DBS was the management of data constituting a database /ANS75/, the new situation calls for systems for multiple databases. The basic property of such databases is that they are typically independently created and administered /HEI85/, /HEW85/, /SEL84/. They present then various logical and physical differences. The logical differences may concern data definition (names and value types), data manipulation languages or even entire data models. In the Annex, we show examples of differences that may occur. The physical level differences may concern data formats, login procedures, concurrency control,... /CER84/, /GLI84/.

One approach to the design of systems for multiple databases is to try to integrate all databases under a single conceptual schema. This schema is called *global schema*. It should define from all data a conceptually single integrated database. Users would then manipulate data as if they constituted a classical database. This approach characterizes for instance /LAN82/.

The idea of global schema was in fact introduced by early distributed database systems : SDD-1, Porel, Sirius-Delta,... (see /CER84/ and /LIT82/ for the corresponding discussion). These systems were designed on the assumption that a distributed database differs from a classical one only through its physical implementation /ROT78/, /CER84/, /FER82/, /ELM84/. The global schema was generally assumed to be relational. The database was assumed to be created from scratch. The distribution was achieved through the splitting of global relations into those on sites (network nodes). The site relations called *fragments* resulted from relational operations /BER80/. This so-called *top-down* design of the distributed database was assumed to be done by the (distributed) database administrator. The administrator was assumed to have total control over data.

However, when the idea of the global schema is applied to multiple databases that preexist their common usage or belong to independent administrators, then the global schema has to be basically designed *bottom-up*. The integration even of only relational databases cannot then in general be performed only

through relational operations. It must also resolve the so-called *semantic conflicts* /CER84/. These conflicts are due to differences between data to integrate with respect to names, values and meanings. The differences result from the fundamental fact that the same reality may be perceived in different ways /RIO82/ . This situation makes the creation of the global schema typically difficult, even for only a few databases /LYN83/, /CER84/, /HEI85/, /LIT85a/. In particular, if databases disagree about a value, then there is no single integrated value satisfactory for all users. Furthermore, no general technique for updates through the global schema seems to exist. Finally, even for organizational reasons alone, a single schema over the thousands of databases on future Open Systems resembles a dream /HEW85/.

A more general approach to the distributed database design may then be to assume that basically there is no global schema /LIT80-81/, /LYN83/, /HEI85/ and /HEW85/. The user will therefore typically face multiple databases with perceptibly distinct schemas. This approach was the basis of the proposals in /LIT82-86/, /HEI85/ and /HEW85-85a/. The system for the management of multiple databases was called *multidatabase (management) system* (MBS) /LIT82/. The term is now quite widespread /BRE84/, /DAY84/, /DAY85/, /LEF84/, /STA84/ ...

The analysis of case studies shows that the design of an MBS should rely upon the following considerations :

- the databases may be *physically distributed* which means that they are at different sites. Site names, connection procedures etc., should then be transparent to the logical level, as the logical and physical levels should be independent (this is in particular the case of actual videotex systems). Databases may also be all at the same site. Such *logically distributed* databases /LIT81/, /HEI85/ will be typical of public database servers.

- the databases may use the same data model as would be the typical case of databases of the same DBS or of a public server. They may also use different data models. In the latter case, it is reasonable to assume that, for the common usage, the databases present themselves according to some common (standard) model (this is already the case of videotex systems and of the European bibliographic databases). The relational model is a particularly important candidate for the future standard. In a few years, most databases will indeed be either relational or at least provided with relational interface. SQL-like interface should be particularly popular, as its standardization is already underway.

- the schemas that the user perceives may be conceptual schemas of centralized databases. They may also be view schemas defined for the common usage (export schemas in /HEI85/) or even some global schemas. For the user, any such schema will correspond to a separate database. On the one hand, the user may then need to manipulate data all within the same database. On the other hand, he may need to manipulate data from different databases. A user may wish for instance to retrieve restaurants and cinemas on the same street from databases **Michelin** and **Cinemas** in the Annex. Or, he may need to rename a street in both databases. Then, he may wish to copy good restaurants from a public database into his personal database, etc.

- some semantic heterogeneity will remain at the common model level. Data designed for the same aims, but in different databases, will frequently differ with respect to names and values. Vice versa, syntactically the same data may correspond to different semantics. This situation will differ from the one within a single database. It will be a counterparts of database administrators (data owners) autonomy /SEL84/. The autonomy should nevertheless be typically appreciated by data owners, as it enables them to privilege their own needs. It should also be appreciated by users, who like to dispose of varieties of views of the reality. It is indeed well known that even an expert view may be only partly true /RIO82/.

Queries to more than one database are called *multidatabase queries* /LIT82/. Data manipulation language (DML) for such queries is called *multidatabase manipulation language* (MML). MML is used at the

standard model level and is the characteristic function of any MBS. Other functions may concern local models translations /LAN82/, /TEM83/, multidatabase views (called import schemas in /HEI85/ and superviews in /MOT81/), interdatabase dependencies /SHI84/, physical distribution /DAY85/, /STA84/...

Functions for multidatabase queries constitute the design challenge of an MML. These functions should especially deal with the semantic heterogeneity. Below, we present such functions within the multidatabase manipulation language MDSL. This SQL-like language characterizes the prototype multidatabase system MRDSM /LIT85a/. MRDSM allows the testing of the design of various functions through their application to the databases of the well known MRDS (Multics Relational Data Store) database system /MUL82/, /RDS83/.

The new functions are intended for simple expression of retrievals and of updates. "Simple" here means that the user intention is expressible through a single (formal) query. The manipulations the MDSL user may perform are basically as follows :

- queries requiring joins of data in different schemas,
- queries broadcasting the user intention over a number of databases with the same or different naming rules for data with similar meanings,
- queries broadcasting the user intention over a number of databases similar in meaning, but with different decomposition into relations,
- dynamic transformation of actual attribute meanings, units of measure etc, into user defined value types,
- queries defining information flow between databases (interdatabase queries),
- various standard (library) functions for dynamic aggregation of data from different databases.

Most of these possibilities are unique to MDSL. Case studies show that they are highly desirable.

The next section presents MDSL. Section 3 concludes the discussion. The Annex discusses the example databases.

2. MDSL LANGUAGE

2.1. An overview

MDSL allows the manipulation of multiple MRDS databases. It extends the classical DML of MRDS, called DSL /MUL82/, /RDS83/. New functions are mainly intended for multidatabase queries. Some functions are designed for query editing, help and multiple schemas displaying (this last function is very useful in the multidatabase environment, as the user frequently needs to acquire the knowledge of a database dynamically). Also, any Multics command may be called from MDSL. Thus text editors may be called, programs may be compiled or executed etc. Auxiliary functions of MDSL are described in /WON84a/.

The main functions are designed to meet following needs :

- **simplicity.** It should be simple for the user to express his query. As for SQL, this requirement means for MDSL, that an intention (informal query) should become a single query /COD82/. The query should furthermore be as short as possible. A multidatabase intention should in particular become a single multidatabase query.

It is this sense of simplicity that is behind the evolution of DSLs and seems to reflect user's wishes well. Relational queries are thus generally simpler than navigational ones, both in the discussed sense and according to the general perception. Then, the universal relation interface /ULL83/ may be even simpler, etc.

- **semantic heterogeneity management.** DMLs are designed for data assumed to be integrated prior to their usage /ANS75/. One goal of the integration is to smooth semantic differences between different users' data. Such differences will in contrast often exist between different databases. Multidatabase manipulations should also remain simple in this environment.

- **interdatabase queries.** Current DMLs consider only the *information flow* /COD71/ between a database and workspaces. In the multidatabase environment, data need to flow also between databases. The popular *downloading* and *uploading* notions are practical expressions of this need. An MML should thus in particular allow to formulate *interdatabase queries.*

Case studies show that information may in particular flow from several source relations into several target relations. The corresponding data will indeed frequently be in several interdependent relations in both source and target databases. Then, the flow may require data name or value conversions and/or target schema modifications. As source and target schemas may differ, incoming data or the target schema may indeed need to be adapted.

The databases manipulated by MDSL are defined using the data definition language of MRDS /MUL82/. In addition, MRDSM provides two specific possibilities :

- the administrator may give to some databases a collective name called *multidatabase name.* For instance, databases **Michelin, Kleber** and **Gault_M** may be collectively named **Rest_guides.** Collective names are popular with database servers. Such names may also simplify the expression of some commands. Otherwise, these commands may require an enumeration of the corresponding databases.

- administrators may declare *interdatabase dependencies.* These are constraints on mutual integrity, privacy etc. /REG83/, /ACH84/. A dependency specific to the multidatabase environment is called *equivalence dependency* /SHI84/, /LIT85/. It links key values identifying the same real object in different databases. It may be useful to declare this dependency for some types of multidatabase queries /LIT85/.

2.2. Query form

The general form of MDSL query is as follows :

open name1 mode1 name2 mode2 ...
-db (abbrev1 name1) (abbrev2 name2) ...
<auxiliary clauses>
-range (tuple_variable relation) ...
-select <target list>
-where <predicates>
-value value_list
<query commands>
close name1 name2 ...

As usual, the **open** command opens the databases for processing. The mode argument specifies the opening mode (for retrieve or update, exclusive or shared). Names may be database names or multidatabase names. The **close** command closes the databases. Both commands are optional if the databases to be used are already open or should remain open for further queries.

The **-db** clause is also optional. It makes it possible to define abbreviations that may be easier to use. It also makes it possible to define the set of the databases the query should refer to, without closing unused ones (to close a database is usually a heavy manipulation). The databases that query refers to are called

scope of the query. Open databases constitute the maximal and default scope.

The *auxiliary clauses* are clauses that do not exist in DSL. They are introduced specifically for the multidatabase environment. The corresponding syntax and semantics will be presented below.

The clauses **-range**, **-select**, **-where** are *main clauses*. Their syntax is basically similar to the one of DSL. The semantics may nevertheless differ even for exactly the same formulation. **-value** clause exists only for updates. "value_list" is a list of new values.

Finally, the query commands are : **retrieve**, **modify**, **store**, **delete**, **copy**, **move** and **replace**. The first four commands are DSL compatible. The last three are used for interdatabase queries. Commands may have parameters or clauses which will be discussed later on.

The names that a query uses for referring to data types are called *designators* /LIT84b/. In DSL and generally in the existing DMLs, designators are unique (unequivocal) identifiers of data types (an attribute, a relation, an entity type, a record type,...). If several attributes bear the same name, then the corresponding designators use relation names as prefixes providing unique identifications. If therefore one calls *designator scope* the set **S** of designated types, then the general rule for relational languages is **card(S)** = 1 for all **S**. This is not always the case in MDSL, where one may have **card(S)** > 1. The corresponding reasons will be shown later on.

2.3. Elementary queries

An MDSL query is an elementary query if all designators are unique identifiers of data types within the scope. The result of an elementary query is a temporary relation, or an update to a database relation. DSL queries, as well as queries formulated using known relational DBSs /RDS83/ are all *elementary monodatabase queries.* An elementary multidatabase query differs from a DSL query by the property that designators may concern relations in different schemas. The **-where** clause of any elementary multidatabase query involves then interdatabase joins. The corresponding implementation issues in MRDSM are presented in /WON84/ and /WON84a/.

The syntax of an elementary query in MDSL may differ from the one in a DSL query. In particular, MDSL makes it possible to use:
- attribute names alone .
- database names as prefixes for unique identification of relations.

In the multidatabase environment, it may indeed happen that two relations in different databases bear the same name.

Example 1 : Retrieve from **My_rest** and **Cinemas** the names of restaurants and of cinemas that are in he same street.

```
-db (m My_rest)(cn  Cinemas)
-range(x  R)(y  cn.C)
-select  x.rname  y.cname
-where  (x.street  =  y.street)
retrieve
```

The designated relation **C** is prefixed in order to distinguish it from the relation **C** within the database **My_rest**.

2.4. Multiple queries

2.4.1. The concept

This function is intended for situations where various databases model the same universe, like that of restaurants in Paris for instance. Case studies show that the user may then need to broadcast the same manipulation to several databases. For instance, the user may wish to project any relation describing restaurants on the attribute expressing the restaurant type.

Present relational languages do not allow to simply express such intentions. If only elementary queries are available, than the user needs to formulate as many queries as there are databases. These queries may furthermore differ from database to database. Multiple queries allow in contrast to broadcast the intention through a single query. This may be a considerable simplification, especially for larger scopes. Alternative names for a multiple query are *diffusion query or broadcast query* /LYN83/.

Formally, a multiple query is a query where some designators designate more than one data type. Basically, these types are in different databases, but they may be in the same database as well. The query is considered as the set of all elementary queries that may result from all choices of unique identifiers within the designators scopes. These queries are called *subqueries.*

It may happen that the choice of unique identifiers leads to a subquery that cannot be executed. We call *pertinent* the executable subqueries. The result of a multiple query is basically the set of relations produced by all and only pertinent queries.

In MDSL, multiple queries are basically formulated through the application of the following new concepts /LIT84b/:

- multiple identifiers,
- semantic variables,
- options on the target list.

2.4.2. Multiple identifiers

A *multiple identifier* is a name shared by several attributes, relations or databases. For instance, if the scope of the query is **Michelin** and **Gault_M**, then the designator **R** is the multiple identifier of both **R** relations and **type** is the multiple identifier of both **type** attributes. A multiple query with multiple identifiers is an equivalent of the set of pertinent subqueries resulting from all the combinations of the unique identifiers in the scope of the multiple ones.

This function is intended for the broadcast of manipulations of data bearing the same names. Syntactically, the queries are basically formulated as elementary queries. However, the meaning of designators that are multiple identifiers is of course different.

Example 2 : (Q1) Retrieve from **Michelin** or **Gault_M** restaurants that are chinese according to a guide.

```
open Michelin Gault_M er
-range (x R)
-select x
-where (x.type = "chinese")
retrieve
```

This query would be the equivalent of two queries :

```
open Michelin er
-range (x R)
-select x
-where (x.type = "chinese")
retrieve

open Gault_M er
-range (x R)
-select x
-where (x.type = "chinese")
retrieve
```

The result would be the set of two relations. Each relation would inherit the database(s) name(s) its attributes come from. The relations would not be union compatible, since their arities and attributes would differ. As guides may disagree upon the type of a restaurant, a restaurant could figure in both or in only one of relations. The location would then be semantically meaningful, as it would implicitly indicate the guide that considers the restaurant as chinese.

Example 3 : Delete from **Michelin** the restaurant with the key **r# = '456'.**

```
open Michelin eu
-select r#
-where (r# = "456")
delete
```

The query would delete the tuples from all relations that have attribute **r#**. It would thus replace three classical relational queries. In addition, it would automatically preserve the referential integrity. This is not the case of queries one may formulate using known relational languages /ULL83/.

2.4.3. Semantic variables

We call *semantic variable* a variable whose domain is data type names. In MDSL the domain may be :
- *explicit*, which means that names are enumerated in an auxiliary clause,
- *implicit*, which means that they result from the variable name.

The aim in this function is to enable the user to broadcast his intention over data named differently. A query may invoke several semantic variables, together with multiple identifiers. Each semantic variable means that the query concerns all the names in its domain. The names may in particular be multiple identifiers. The query is equivalent to the set of pertinent subqueries resulting from possible substitutions of semantic variables and multiple identifiers by unique identifiers.

Explicit domains

The corresponding clause is :

-range_s $(x^1.x^2. \ldots .x^k \quad n^{1,1}.n^{2,1}. \ldots .n^{k,1} \quad n^{1,2}.n^{2,2}. \ldots .n^{k,2} \ \ldots)\ldots$

Each **x** is a semantic variable. Each **n** is a name. i-th subquery corresponds to the simultaneous substitutions of $n^{j,i}$ to x^j ; j=1,...,k .

Example 4 : (Q2) Retrieve from **Rest_guides**, i. e. from **Michelin, Kleber** and **Gault_M** databases, restaurants that a guide considers as chinese.

```
open Rest_guides r
-range_s (x R REST)
-range(y x)
-select y
-where (y.type="chinese")
retrieve
```

X is a semantic variable whose values are names **R** and **REST**. Since **REST** is a unique identifier, the corresponding substitution produces an elementary query. In contrast, since **R** is a multiple identifier, it leads to two elementary queries, equivalent together to (Q1). All three resulting relations are not union compatible. As for (Q1), they may also contain different restaurants.

Example 5 : Retrieve from **Cinemas** and from **My_rest** the restaurants and the cinemas that cost less than 30 Ff.

```
-db (cc Cinemas)(mr My_rest)
-range_s (x.y.y#     R.M.m#      C.P.p#)
-range (z x)  (v y)
-select z
-where (z.y# = v.y#)&(v.price < 30)
retrieve
```

The query will lead to two differently formulated subqueries, one per database in the scope.

Example 6 : Change to "123" the phone number "876" in all example databases.

```
open Rest_guides Cinemas My_rest eu
-range_s ( t   tel   t#)
-select t
-where (t = "876")
-value "123"
modify
```

The query will replace five elementary queries. It could in fact replace any number of such queries, provided the database owners agree to name telephone either **tel** or **t#**. Thus it is not even necessary for all administrators to agree upon a common name.

Implicit domains

Here a variable name contains one or more special characters that at present are :

- '*' designating any string of digits, including the empty string,
- '?' designating any but only one digit,

The domain of the variable is then implicitly constituted from all names in the scope that match the resulting pattern. For instance the variable x such that x = 'R*', will lead to all names starting with R. The subqueries correspond to all pertinent substitutions of data names within the domains.

If the characters '*' and '?' are parts of data names, as in data named **'R*'** for instance, then they should be preceded by the character '\'. This character means "escape" within Multics system. Thus **x = 'R*'** would include all names starting with the string **'R*'**.

Example 7 : The expression of query (Q2) from Ex. 4 may be simplified to the following one :

```
open Rest_guides er
-range(y R*)
-select y
-where (y.type="chinese")
retrieve
```

Again this formulation would remain valid for any number of databases, provided the databases agree that all and only restaurant relation names start with the character **R**. Except for this constraint, they may be called by any name best suited to the administrator's own needs : **R, Rest, Restaurant,...**

2.5. Options

Current relational DMLs assume implicitly that all the attributes in the **-select** clause target list are required. Case studies show that this assumption should be relaxed in the multidatabase environment. The concept of *options* /LIT84b/ is intended for this purpose. The corresponding syntax is as follows.

Let **d** be an attribute designator within **-select** clause. Let **q** be a subquery resulting from some substitutions and **a** the unique identifier corresponding to **d** in **q**.
-if **d** is preceded by space, as is usual in DSL, then **q** is not pertinent if there is no attribute **a** in its scope. Thus, by default **a** is mandatory.

- **d** written '~d' means that **q** may be pertinent without an attribute named **a** in the scope. **q** is then considered as equivalent to a query formulated like **q** without **a** in the **-select** list. The attribute **a** is thus optional.

- a list $d_1|d_2|\ ...\ |d_n$ means that the pertinent form of **q** should contain one and only one a_i. The choice follows the list order. A list preceded with '~' means that the whole list is optional.

Options deal with the existence of attribute names in schemas and not with null values within tuples. However, one may extend this concept to null values as well.

Example 8 : Retrieve from **Rest_guides** name, street and owner, if any, of all restaurants.

```
open Rest-guides er
-range(x R*)
-select x.*name  x.street ~x.owner
retrieve
```

Since the attribute **owner** is optional, all three databases will be addressed. If **owner** was mandatory, the tuples would be retrieved only from **Kleber** database.

Example 9 : Assume that **Gault_M** does not have the attribute **tel**. Retrieve from **Rest_guides** restaurant names and either phone numbers if available else the corresponding streets.

```
open Rest-guides er
```

```
-range  (x  R*)
-select   x.*name,  x.t*|x.street
retrieve
```

The query will provide the telephone number from **Michelin** and **Kleber**, and the address from **Gault_M**.

2.6. Incomplete queries

2.6.1. The concept

While formulating MDSL queries, the user may avoid specifying some equijoins. Basically, one may avoid equijoins linking primary or foreign keys, that share a domain. Such queries are called *incomplete queries.* A subquery of a multiple query may in particular be an incomplete query. Omitted joins are called *implicit joins* /LIT85/. They are deduced by the system from database schemas. The result is called *complete query.*

This function has a double goal :

- to further simplify query formulation,
- to allow multiple queries to databases modeling the same universe, but different through decompositions into relations.

There is indeed sometimes no way to express an intention in a single query, if one has to formulate all equijoins corresponding to different decompositions.

Example 10 : Retrieve from **Michelin** the address of all restaurants that serve "confit d'oie".

The incomplete query could be :

```
open Michelin er
-select street                                                    (Q3)
-where  (cname = "confit d'oie")
retrieve
```

The complete query would be :

```
open Michelin er
-range(x R)(y  M)(z C)                                            (Q3')
-select  x.street
-where (z.cname = "confit d'oie") &(x.r# = y.r#)&(y.c# = z.c#)
retrieve
```

Example 11 : Consider now that instead of three relations **Gault_M** contains only one (universal) relation with all attributes in **Gault_M**. Assume further that the user wishes to broadcast the query about "confit d'oie" to both **Michelin** and **Gault_M**. The formulation (Q3) will then remain valid, provided both databases are open. The clauses will however define a multiple query. The query will be the equivalent of two subqueries differing by equijoins. These are (Q3') and the query :

```
open Kleber er
-select street
-where  (cname = "confit d'oie")
```

retrieve

Example 12 : Update queries may be incomplete as well. The query : "delete from **My_rest** all the courses whose **ncal** > 2 000" may be formulated as follows :

```
open  My_rest eu
-select C
-where  (ncal > 2 000)
delete
```

2.6.2. Completion principles

The completion process is described in detail in /LIT85/. The basic case is the one of an incomplete elementary conjunctive query. Other types of incomplete queries, like incomplete disjunctive queries or set type queries or multiple queries, are completed through the iteration of the basic algorithm for each of their conjunctive components. Depending on the type of query and on the set operators used, the result is either the set or the union or the intersection etc. of the resulting complete subqueries. Below, we shall focus on the basic case.

The (multi)schema is considered as a graph, called *(multi)database graph,* whose nodes are relations and arcs are connections through key attributes sharing a domain. It may also include interdatabase arcs, if equivalence dependencies are declared. The query is also considered as a graph, called *query graph,* whose nodes are the relations addressed in the query and the arcs are join clauses. Disjunctive, set type or multiple queries are represented by sets of such graphs corresponding each to one conjunctive components. The query is incomplete iff one of its graphs is disconnected.

The first step of the completion algorithm is to find within the (multi)database graph all the minimal connected trees that include the incomplete query graph. Each tree is then considered as representing a conjunctive complete query whose join clauses correspond to the tree arcs. The implicit joins correspond then to the arcs that had to be added to the original graph in order to render it connected. If several minimal trees exist, the corresponding subqueries are unioned.

Examples above and in /LIT85/ show that this process usually leads to the intuitively expected result. If not, the user may explicit the joins which were misunderstood or may ask for interpretations including non minimal trees as well.

The goal of the concept of incomplete query is to some extent similar to the one of the concept of the universal relation /ULL83/, KEN83/. However, MDSL does not assume that the schema should be seen as a single relation. Corresponding consequences are discussed in /LIT85/. A major consequence is that updates may be performed as well.

2.7. Dynamic attributes

Dynamic attributes are transforms of actual attributes. They are dynamically defined within a query and unknown to any schema. Except for eventual update limitations, they may be manipulated as the actual attributes /LIT86/.

The aim in this function is to allow the user to dynamically and subjectively transform data values. Such a need will be frequent in the multidatabase environment. The function will be in particular frequently necessary for interdatabase joins. Joins are indeed meaningful only if the involved data have the same meaning and unit of measure.

The MDSL user may declare a dynamic attribute by means of the following auxiliary clauses :

-attr_d [hold] a : C/R
-define by MT(s) = m_j
-updating s' by MT(s") = m'
a is the dynamic attribute name with value type either **C** (character) or **R** (real). If there is no **hold** argument, then **a** is known only within the query that defines it. Otherwise, further queries from the user may also refer to **a**, until the end of the session.

The clause **-define by** defines the mapping **m** of actual attribute(s) **s** on **a**. It is mandatory for retrievals. **MT** denotes the mapping type. It may be **D** for a dynamically defined dictionary (table), **F** for a formula or **P** for a program. The corresponding clause forms are respectively as follows :

- **define by D(s) = (a_1, s_1),...,(a_k, s_k)**
- **define by F(s) = arithmetical_formula**
- **define by P(s) = Multics_segment_name**

s_j are actual values and a_j the corresponding dynamic ones. Formulas are arithmetical formulas. The Multics segment contains the program that may be written in any programming language.

The clause "**-updating**" defines the mapping of **a** on an actual attribute **s'**, which is needed when the user updates **a** . The attribute **s'** should be one of the actual attribute(s) in **s**, and **s"** are all the other attributes in **s**, if they are any. This clause is currently mandatory if **MT** in **-define by** clause is **P** or **F**. It is optional for **D** type mappings. The default option is then that a given **a** value, let it be **a'**, corresponds to the first s_j such that $a'=a_j$. In all cases, mapping types in both clauses have to be the same.

A dynamic attribute may share the name of an actual one. If some of the actual attributes defining **a** are not in the scope of the (sub)query, the name in the (sub)query designates then the actual attribute. Otherwise, it designates the dynamic attribute.

The user may also wish to refer to an actual attribute **n** that shares the name of a dynamic one, previously defined using **hold** argument. Then, the **-select** clause has to be preceded by the clause **-actual n**.

Example 13 : Assume that "********" of **Michelin** corresponds to **Gault_M.qual** > 16/20. Retrieve from **Michelin** and from **Gault_M** restaurants rated "********".

```
open Michelin Gault_M er
-range (t R)
-attr_d stars : C
-define_by   P(qual) = star
-select t
-where (t.stars = "****")
retrieve
```

This query leads to two subqueries. The first one to **Michelin** will select the actual attribute, since the attribute **qual,** used for the definition of the dynamic attribute **star,** is not in this subquery scope. The second subquery will produce the values of the dynamic attribute and will use these values for **-where** clause evaluation with respect to **Gault_M**. The overall result of the query will be homogenized with respect to the **Michelin** scale of rating, arbitrarily transposed by the user to **Gault_M** database. Note

that there is no objective integration rule for **Michelin** and **Gault_M** scales or for subjective scales in autonomous databases in general.

star is the program that computes dynamically through the Multics **execute** command the values of **stars**. It expresses, in an arbitrary host language, the algorithm :

if qual > 16/20 : stars = "***" endif;

The same mapping could also be formulated using the **D** type declaration as follows :

-define_by T(qual) = (***,20), (***,19), (***,18), (***,17)

Note that the mapping is undefined for values of no interest for the query, "**" for instance.

Example 14 : Retrieve from **Michelin** restaurants that have the same average price in **Michelin** and **Gault_M.**

```
-db (m Michelin) (g Gault_M)
-range (t m.R) (v g.R)
-attr_d  price : R
-define_by  F(m.r.avprice) = m.r.avprice * 1.15
-select  t
-where (t.price = v.avprice) &(t.name = v.name) &(t.street = v.street)
retrieve
```

The function renders here the interdatabase join on price meaningful, as the meanings of the concept of price differ in both databases. The clauses referring to **name** and **street** may be implicit, if the corresponding equivalence dependency was defined. More examples of dynamic attributes as well as the discussion of their implementation in MRDSM are in /LIT86/.

2.8. Interdatabase queries

The general form of interdatabase queries is as follows :

copy / move
< source selection expression >
store / modify / replace
-target <db_name>.[<relation_name>]
<mapping clauses>

The commands **copy** and **move** define the action on the source database(s). The **copy** command copies source data, according to the source selection expression, while the **move** command also deletes the source data. Its selection expression has then to designate all attributes of a relation. In both cases, if data values are to be converted, the source selection expression should contain the definition of appropriate dynamic attributes. The meaning of these attributes should be that of the corresponding target attributes. Value type conversions, like that of integers to reals, are automatic.

The commands **store**, **modify** and **replace** define the action on the target. The clause **-target** identifies the target database or relation. The mapping clauses define the matching of the incoming attributes to the corresponding target attributes. The syntax of the mapping clauses is as follows :

<mapping clauses> := **-mapping** [<rule>][<matching_list>]

<rule> := **by order / by name**
<matching_list> := (<option> [, <option>])
<option> := source_name --> target_name / target_name /
source_name --> **'new'**

The source names are the attribute names within the source **-select** clause. The rule **by name** means that source attributes should be mapped on target attributes with the same names, except eventually for the attributes within the matching list. This rule is assumed by default, in particular for the target relation name. The rule **by order** means in contrast that the attributes should be matched in order of their enumeration in the source **-select** clause, on the successive attributes within either the target schema or within the matching list. In the latter case, the elements in the matching list must be only the target names.

The matching list alone specifies an arbitrary correspondence. In particular, the option **'new'** means that the source attribute does not exist in the target and should be added to the target schema. For security reasons, source attributes without the target counterparts and not declared as 'new' are disregarded. Inversely, if a target attribute has no source counterpart, then the corresponding values are set to null or are preserved, depending on the command. Finally, except for the **replace** command, the query is assumed valid only when the key attributes of the incoming relations correspond to the key attributes of the target relations.

The **store** command inserts tuples that do not share key values of existing target tuples and preserves those target tuples that share incoming key values. The **modify** command also inserts incoming tuples without target key counterparts, but it modifies target tuples that share incoming key values. The modification concerns only the attributes that have counterparts within incoming tuples. Neither command affects target tuples whose keys do not share incoming key values. In contrast, **replace** command replaces the whole content of the target with the incoming one.

Example 15 : Copy to **My_rest** restaurants considered as good by **Gault-M,** as well as the associated courses and menus.

```
copy
-db (g  Gault_M) (m My_rest)
-range_s (x g.R  g.C  g.M)
-range (t x)
-select  t
-where  qual > 14/20
store
-target m
```

The **copy** command will produce three subqueries. Two of them will require completion of implicit joins. The whole query will copy three relations, containing respectively the selected restaurants, courses and menus. The result will automatically preserve the referential integrity. The selected relations and attributes will be mapped on those with the same names within the target. Only tuples that do not share key values already in **My_rest** will be stored. Thus the user opinion about a restaurant, a course or a menu, will have the priority. The inverse effect would appear if **modify** command was used.

This query represents the case we spoke about in Section 2.1, where source data in several relations should enter several target relations. The principles of relational model will frequently lead to this case. Think about a supplier and his parts, a student with his courses, a customer and his accounts, etc.

Example 16 : Replace the content of **My_rest** with the restaurants, the related courses and menus that correspond to '**' rating in **Michelin**. Convert the meaning of the **Michelin** prices to those with tip

included.

```
copy
-db (m  Michelin) (my My_rest)
-range_s (x m.R  m.C  m.M)
-attr_d price : R
-define_by   F(m.r.*price) = m.r.*price*1.15
-range (t x)
-select  t
-where  stars = '**'
replace
-target my
-mapping by name (stars-->qual)
```

Example 17 : Consider that the user has changed the schema of the relation **C** into the following one:

C (c#*, origin, cal, name),

where the new attribute **origin** denotes the region or country the course(dish) comes from, if any. The query "copy to **My_rest** the courses in **Gault-M**" may be expressed as follows :

```
copy
-db (g  Gault_M) (m My_rest)
-range (x g.c)
-select  x
store
-target m
-mapping in order(c#, name,  cal)
```

The values of **origin** will be null.

Example 18 : Consider that the user wishes to keep in **My_rest** only the best restaurants (those rated more than 16/20). However, he also wishes to save in a separate database, let it be **My_rest_archives**, the content of relation **R**. The corresponding query may be as follows :

```
move
-db (m My_rest) (a My_rest_archives)
-select  m.R
-where  qual < 17/20
store
-target a
```

2.9. Standard functions

Standard functions may be declared in MDSL in two ways :
- inside the clauses, being then enclosed within square brackets. The function is then evaluated independently for each subquery.
- as independent clauses. The function applies then to all the tuples of the query.

Some functions may be applied only to subqueries, some only have meaning as independent clauses and some may be applied in both manners.

Example 19 : Retrieve average price of meals according to each **Rest_guides** database.

```
open Rest_guides er
-select [avg(price)]
retrieve
```

Example 20 : Retrieve average price of all meals within **Rest_guides** databases.

```
open Rest_guides er
-avg
-select price
retrieve
```

Case studies have shown that in addition to the well known standard functions, the user will need new functions, specific to the multidatabase environment. The following functions should prove particularly useful.

name function

Let **n** be a designator. **name(n)** provides the name of data designated as **n**. **name(.n)** provides the name of the container of data designated by **n** : relation for an attribute etc... Then :

name(..n) = name(.name(.n))

etc.

This function results from the need for relational operations not only on data values, but also on data names. It may be applied instead of an attribute name within **-select** and **-where** clauses. The result is then considered as if it had been an actual attribute value.

Example 21: Retrieve the names of the chinese restaurants and of the guides that recommend them.

```
open Rest_guides er
-range(x R*)
-select  x.*name, x.[name(.R*)]
-where (x.type = "chinese")
retrieve
```

Each tuple will then contain the name of a restaurant and the name of the database the tuple comes from. As may be seen, in the multidatabase environment, database names may bear logical information.

norm function

This function merges into one tuple all tuples corresponding to the same real object. The correspondence results from the equality of the keys indicated by the user. Keys may be the primary keys or the candidate keys, if the values of the primary keys within different databases differ. **norm** function has the following syntax :

-norm ((id_att) tuple_var).

id_att are designators of keys identifying the same object. The function appears only as an independent clause. The result is the natural outer join of the designated relations.

Example 22 : Retrieve from Rest_guides the chinese restaurants, creating one tuple per restaurant.

```
open Rest_guides er
-range(x R*)
-norm((*name, street)  x)
-select x
-where (x.type = "chinese")
retrieve
```

The result would be a single relation **R** whose attributes would be all those of involved relations, prefixed with the database name in the case of name conflict. The id_att would figure only once, according to the definition of natural joins. To any restaurant would correspond exactly one tuple. The values of the attributes corresponding to databases that do not recommend the restaurant would be null. Absence of null values in at least some columns corresponding to attributes from a database would thus be an implicit indication that the corresponding guide recommends the restaurant.

Outer joins are unknown to MRDS. Algorithms for efficient processing of such operations have therefore been investigated /ABD84/. Discussion of such algorithms may also be found in /DAY84-85/.

upto function

This function appears only as an independent clause. It limits the multiplicity of information that may come from several databases. For instance, a query to ten restaurant databases may ask for at most two recommendations of a restaurant. In particular the user may give priority to databases he trusts more than others. The function syntax is as follows :

-upto (n (A) [B]).

The function provides at most **n** ≥ **1** tuples sharing the values of attributes designated in the list **A**. Priorities correspond to the order of the list **B** that designates database names. **A, n** and **B** are optional. If **A** is not specified, the query processing stops after a non null response from **n** databases. The default value of **n** is 1. Finally, empty **B** means that the user has no preference.

Example 23 : Retrieve the number of calories of "confit d'oie", preferably according to **Kleber.**

```
open Rest_guides er
-upto (1  (cname) [Kleber])
-select  ncal
-where (cname = "confit d'oie")
retrieve
```

The result will come from another database only if **Kleber** does not describe confit d'oie.

Example 24 : Retrieve all the chinese restaurants from **Rest_guides**, but limit the number of a restaurant's descriptions to two. Give the priority to **Michelin** and **Kleber** descriptions.

```
open Rest_guides er
-range(t  R*)
```

```
-upto (2  (t.*name t.street)  [Michelin Kleber])
-select t
-where(t.type = "chinese")
retrieve
```

A restaurant will figure in the output from **Gault_M** only if it was not in the output from **Michelin** or **Kleber**.

3. CONCLUSION

As databases become easily accessible on computers and networks, more and more users face multiple databases. New functions for data manipulation languages are then needed, as the present languages were designed for manipulations of a single database. These functions should especially allow users to manipulate autonomous databases. MDSL offers some corresponding functions for relational databases. Most future distributed databases will be of this type or will at least present a relational view.

The MDSL functions were designed for user needs that appeared from case studies. Numerous examples showed that they should prove useful for a large variety of such needs. In particular, they should be frequently more practical than the global schema, when one may be constructed. This is due to their flexibility and open nature with respect to the accommodation of autonomous names, values and structures. Most of these functions are not yet known to other languages and systems.

Although the functions are designed for MRDS databases, their principles are data model independent. They may thus be easily transposed to other relational languages. The concepts of of multiple identifiers, of semantic variables, of interdatabase queries etc. may furthermore be applied to other data models. They may thus be a useful basis for the design of distributed systems using also other popular models /MOU81/.

In particular, several functions have also proved useful in a single database context. Thus, the concept of the multiple identifier may help in preserving the referential integrity. Implicit joins simplify the formulation of most relational queries. Dynamic attributes are useful for applications where subjective or frequently changing value mapping rules render the traditional concept of view too static. It should thus be worthwhile to incorporate similar functions to any relational system.

New functions lead to many interesting problems at the implementation level that are discussed in the corresponding papers. Many issues relative to performance optimization are still open. On the other hand, logical possibilities of current functions may also be enhanced and more functions are needed. Knowledge processing techniques should prove particularly useful for both purposes /HEW85a/, /LIT86/. First, they should enlarge the class of intentions expressible as a single query. Then, they should make it possible to further simplify the expression of some queries. For instance, it should be possible to frequently avoid the **-updating** clause which is currently usually mandatory for a dynamic attribute update. The update mapping being an inversion of that defined for retrievals, expert systems may indeed be taught to deduce many such inversions automatically.

REFERENCES

/ABD83/ Abdellatif, A. Multiple queries in the multidatabase system MRDSM (in French). Th., Univ. of Tunis. INRIA, (June 83), 80.

/ABD84/ Abdellatif, A. External joins and standard functions in the multidatbase system MRRDSM (in French). Rapp. DEA, Univ. Paris 6, (June 1984), 67.

/ACH84/ Achour, M. Privacy in the multidatabase system MRDSM (in French). Th. University of Tunis, INRIA, (June 1984), 216.

/ANS75/ ANSI-SPARC interim report on database management system. Doc. 7514TS01, Washington DC, (Feb. 1975).
/BER80/ Berstein, P., A. et al. Query processing in a System for Distributed Databases (SDD-1). ACM-TODS, 6, 4, (June 1980).
/BRE84/ Breitbart, M. ADDS : Another multidatabase management system. Distributed Data Sharing Systems. North-Holland, 1985, 7-24
/CER84/ Ceri, S. Pelagatti, G. DISTRIBUTED DATA BASES Principles and Systems, McGraw-Hill Book Company, 1984, 393.
/COD71/ Codd, E., F. A database sublanguage founded on the relational calculus. ACM SIGFIDET, (Nov. 1971), 35-68.
/COD81/ Codd, E., F. SQL/DS : what it means. Computerworld, (Feb 1981), 27-30.
/COD82/ Codd, E., F. Relational Database: A Practical Foundation for Productivity. CACM, 25, 2 (Feb 1982).
/COR84/ Cortes, R. MRDSM: optimisation of processing of multidatabase queries. 3-rd Int. Sem. on Distr. Data Sharing Systems, Parme, (March 1984), University of Parma, 40-42.
/DAY84/ Dayal,U., Gouda, M., G. Using Semiouternjoins to Process Queries in Multidatabase Systems. ACM-PODS, Waterloo (Canada), (Apr. 1984), 153-162.
/DAY85/ Dayal, U. Query Processing in a Multidatabase System. ON CONCEPTUAL MODELING. Springer-Verlag, 1985, 81-108.
/ELM84/ Elmasri, R. and S Navathe, Object Integration in Logical Database Design, Proceeding of the International Conference on Data Ingineering, Los Angeles, (April 1984), 426-433.
/FER82/ Ferrier, A and C. Stangret, Heterogeneity in the Distributed Database Management Systems Sirius-Delta, Proceeding of the 8th VLDB International Conference, Mexico City, September, 1982
/GLI84/ Gligor, V., D. Popescu, R. Concurrency Control Issues in Distributed Heterogeneous Database. Distributed Data Sharing Systems. North-Holland, 1985, 43-56.
/HEI85/ Heimbigner, D. McLeod, D. A Federated Architecture for Information Management. ACM-TOIS, 3, 3, 1985, 253-278.
/HEW85/ Hewitt, C., de Jong, P. Open Systems. ON CONCEPTUAL MODELING. Springer-Verlag, 1985, 147-164.
/HEW85a/ Hewitt, C. The Challenge of Open Systems. BYTE, (Apr. 1985), 223.
/KEN83/ Kent, W. The Universal Relation Revisited. TODS 8, 4, (Dec. 83), 644-648.
/LAN82/ Landers, T., Rosenberg, R. L. An overview of MULTIBASE. 2-nd Symp. on Distr. Data bases. Berlin, (Sep. 1982). Proc. North-Holland, 1982, 153-184.
/LARS86/ Larson, A. Rahimi, S. Distributed Database Management Systems (Tutorial) IEEE-COMPDEC, Los Angeles, (Mai 1984).
/LEF84/ Lefons, E. Silvestri, A. Multidatabase systems. DISTRIBUTED DATA SHARING SYSTEMS. North-Holland, 1985, 25-42.
/LIT80/ Litwin, W. A model for a distributed data base. 2-nd ACM Comp. Exp. University of Laffayette. Luisiane, (Feb. 1980), 54-71.
/LIT81/ Litwin, W. Logical model of a distributed database. DISTRIBUTED DATA SHARING SYSTEMS. North-Holland, 1982, 173-207.
/LIT82/ Litwin W. et al. SIRIUS Systems for Distributed Data Management. DISTRIBUTED DATA BASES. North-Holland, 1982, 311-366.
/LIT83/ Litwin, W., Kabbaj, K. Multidatabase Management Systems. ICS 83, Nurnberg,(March 83). B. G. Teubner, 482-505.
/LIT84a/ Litwin, W. MALPHA : a relational multidatabase manipulation language. IEEE-COMPDEC, Los Angeles, (Mai 1984), 234-242.
/LIT84b/ Litwin, W. Concepts for multidatabase manipulation languages. JCIT-4, Jerusalem, (June 1984), 433,442.
/LIT85/ Litwin, W. Implicit joins in the multidatabase system MRDSM. IEEE-COMPSAC, Chicago, (Oct. 1985).
/LIT85a/ Litwin, W. An overview of the multidatabase system MRDSM. ACM Nat. Conf, Denver, (Oct. 1985).

/LIT86/ Litwin, W., Vigier, Ph. Dynamic attributes in the multidatabase system MRDSM. IEEE-COMPDEC-2, Los Angeles (Oct 1986), 8.
/LYN83/ Lyngbaek, P., McLeod, D. An Approach to Object Sharing in Distributed Database Systems. VLDB 83, Florence, (Oct. 1983), 364-376.
/MOT81/ Motro, A., Buneman, P. Constructing Superviews. ACM-SIGMOD 1981, 56-64.
/MOU81/ Moulinoux, C., Faure, J. C., Litwin, W. MESSIDOR system. ACM-SIGSMALL Symposium on Small Systems. Florida. (Oct 1981) 130-135.
/MUL82/ Multics Relational Data Store (MRDS) Reference Manual. CII-Honeywell Bull. Ref. 68 A2 AW53 REV4, (Jan. 1982).
/RDS83/ Relational Database Systems. Schmidt, J., W., Brodie, M. (ed.). Springer-Verlag, 1983, 618.
/REG83/ Regaieg, N. Manipulation complements in the multidatbase system MRDSM (in French). Thesis Univ. of Tunis. INRIA, (June 83). 67.
/RIO82/ Riordan, J. (ed.) Six blinds and an elephant. A WORLD OF FOLK TALES. Hamlyn, London, 1982, 56-59.
/ROT78/ Rothnie, J. B. Distributed database management. VLDB-78, Berlin, (Sep. 1978).
/SEL84/ Selinger, P. et al. The impact of site autonomy on R*. Databases-Role and Structure. Cambridge Univ. Press, 1984, 151-176.
/SHI84/ Shili, B. Interdatabase dependencies and Implicit joins in the multidatabase system MRDSM (in French). Thesis, Univ. of Tunis. INRIA, (June 1984), 216.
/STA84/ Staniszkis, W. et al. Network database management system architecture. DISTRIBUTED DATA SHARING SYSTEMS. North-Holland, 1985, 57-76.
/TEM83/ Templeton, M., D. Brill, A. Hwang, I. Kameny, E. Lund, An Overview of the Mermaid System-- A Frontend to Heterogeneous Databases, Proceeding of EASCON 83, September 1983.
/ULL83/ Ullman, J., D. Principles of Database Systems. 2-nd ed. Computer Science Press.
/WIE83/ Wiederhold, G. Database design. McGraw-hill Book Company, 1983.
/WON84/ Wong, K. Design and Implementation of the Relational Multidatabase System MRDSM. Dr. Thesis, Univ. Paul Sabatier, Toulouse (June 1984), 229.
/WON84a/ Wong, K. Bazex, P. MRDSM : a relational multidatabase management system. Distributed Data Sharing Systems. North-Holland, 1985, 77-87.

ANNEX

The schemas that follow define the databases we use throughout the examples. The relations are defined according to MRDS data definition language. We avoided the domain and attribute declarations, mandatory for actual MRDS schemas. The character "*" identifies key attributes.

DB Cinemas:

C (c#*, cname, street, tel),	**Cinemas**
M (m#*, mname, kind),	**Movies**
P (c#*, m#*, hour, price);	**Projections**

DB Michelin:

R (r#*, rname, street, type, stars, avprice, tel),	**Restaurants**
C (c#*, cname),	**Courses**
M (r#*, c#*, price);	**Menus**

DB Kleber

REST (rest#*, name, street, type, forks, t#, owner, meanprice),
C (c#*, cname, ncal),
MENU (rest#*, c#*, price);

DB Gault_M:

R (r#*, rname, street, qual, tel, type, avprice),
C (c#*, cname, ncal),
M (r#*, c#*, price);

DB My_rest:

R (r#*, rname, street, qual, tel, type, avprice),
C (c#*, cname, ncal),
M (r#*, c#*, price);

The schemas model actual applications, essentially on the French public videotex system **Teletel**. The database **Cinemas** models a public database describing the current cinema programs in a city. **Michelin, Kleber** and **Gault_M** model public databases defined upon famous French restaurant guides with the same names (the full name for **Gault_M** is **Gault-Millau**). **My_rest** is a personal database in which a user stores the restaurants of his choice, using as a reference the **Gault_M** model and data. Some of **My_rest** restaurants may nevertheless be unknown to **Gault_M**. Then, some of the restaurants in both databases may be characterized by different values. This would mean that the user replaced in **My_rest** the **Gault_M** opinion about a restaurant his own. Of course he could not do it if he did not have his own database.

The relations within the restaurant databases model respectively restaurants, courses (dishes) and menus. The attributes are based upon the actual guides. All data are data model homogeneous, as all databases are relational. However, data are to some extent semantically heterogeneous. This is because of the following properties, modeling the actual ones and due to the data bases' autonomy :

1. Guides partly disagree upon (i) the choice of attributes that should model the universe of restaurants (ii) the names modeling the same concepts.

2. A restaurant may be recommended by more than one guide, but not all restaurants are recommended by all the guides. The situation is similar with respect to courses.

3. Despite the same names, primary key values modeling the same object in different databases are independent.

4. The units and scales of restaurant quality ratings differ from guide to guide. **Michelin** rates restaurants from none to three stars ("***"). **Kleber** ratings are from zero to four "forks". **Gault_M** rating is m/20 ; m = 1, ..., 20. There is no objective specific rule for ratings correspondence. Nevertheless, it is frequently clear that guides disagree upon a restaurant.

5. The guides may also disagree upon the average price of a meal or upon a restaurant type. For instance, a restaurant may be chinese for one guide and vietnamese for another. The guides may disagree also upon the phone number, although it is a candidate key within each database.

6. In particular **Michelin** and **Gault_M** disagree upon the meaning of attributes dealing with prices, despite the same attribute names. **Michelin** prices are without the 15 % tip, mandatory in France, while **Gault_M** prices are tip included.

7. In contrast, the guides always agree upon a restaurant name and the corresponding street name. This property thus identifies the same restaurant in different guides. An analogous situation exists with respect to course (dish) names.

Similar properties will be typical of the whole multidatabase environment. Multiple databases modeling the same universe such as the travel industry or banking or insurance etc., will resemble each other, but will also present numerous semantic differences.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique