Multidatabase Interoperability

Witold Litwin and Abdelaziz Abdellatif
Institut National de Récherche en Informatique et en Automatique

Many users now have an interest in simultaneoulsy accessing several databases. We present the main features of a prototype relational system designed specifically for this purpose.

The development of database systems, or 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, Teletel, Telidon, etc., provide hundreds of databases on almost any subject such as cinema, train, and airline schedules; banking services; and restaurant fare. An increasing number of users have an interest in simultaneously accessing and manipulating data from several databases. A user may search for restaurants through several restaurant guides or may check several airlines for the cheapest flight, or may need to extract data from a public database for his personal database, etc.

The basic property of such databases is that they are independently created and administered.[1-3] Since each administrator of a database has his own database needs, databases differ physically and logically. The physical differences may concern data formats, login procedures, concurrency control, etc.[4] The logical differences may concern data manipulation languages or even entire data models. Even if the participating databases all use the same data model, they usually present mutual semantic conflicts.[4] These conflicts are differences, redundancies, or incompatibilities with respect to names, values, and meanings among similar data. They result from different perceptions of the same reality by different people. In the sidebar on page 13, we show examples of differences that may occur.

This situation calls for a new type of system designed to manage multiple databases. Such systems have been called multidatabase (management) systems or MBSs[5]—a term now rather widespread. One may attempt to base the design of such a system on the idea of global schema. This schema should define from all databases a logically single, integrated database. Users should then manipulate only data of the global schema or of an external schema derived from it. In both cases, they should feel as if they were in front of a classical database for which the global schema would constitute the classical conceptual schema. This is the approach taken, for instance, in MULTIBASE.[6]

However, it appears that the creation of a global schema is usually difficult.[1,4,7,8] This is the case even if the participating databases constitute only a small number and present the same data model for the common usage. The main reason is the lack of a general solution for the semantic conflicts in a situation in which the autonomy of each of the constituent databases is preserved. In particular, if the 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 for the thousands of databases on future open systems is a dream.[2]

A more general approach may be to assume that the databases the user may access basically have no global schema.[1,2,7,9] The user will then in general know that he faces multiple databases. The system should provide him with functions for manipulating data that may be in visibly distinct schemas and may be mutually nonintegrated. One may say that the con-

0018-9162/86/1200-0010$01.00 © 1986 IEEE

stituent databases would then become interoperable, instead of being manipulable only separately or only as if they were components of a single global database.

Interoperability looks appealing for several reasons. At first, it may of course facilitate manipulation of multiple databases when a global schema does not exist. On the other hand, one still may use such a schema when it may be specified (as the schema of a particular view of the entire collection of interoperable databases[5]). Furthermore, because interoperability does not require integration, it should be convenient for administrators, who usually like to remain autonomous (this is often the reason why they choose to create separate databases). Finally, interoperability should also be appreciated by users, who usually like to face a variety of views of reality of, for example, the ratings or opinions given by several independent restaurant guides.

In this article we present a prototype multidatabase system, called Multics Relational Data Store Multidatabase or MRDSM, that is representative of the interoperable approach. The system renders interoperable the databases that are relational or present a relational view for the common usage. The use of the relational model as the common one seems a reasonable choice, as most of the future databases are expected to be of this type or are expected to be provided with the relational interface. The testbed databases are those of the well-known MRDS, or Multics Relational Data Store, database system.[10] For MRDS, MRDSM is a user among others. In particular, databases manipulable through MRDSM remain accessible directly through MRDS. Also no change to the MRDS system was introduced. The functions for the translation of other data models into the relational one have not been studied. See for instance Brodie et al.[11] for several papers on these issues.

The functions for multidatabase interoperability that MRDSM proposes at present are mainly of two types:

- The administrators of the databases who are cooperating in such a system may define, at the data definition level, names for collections of databases and interdatabase dependencies. The dependencies link the schemas provided for the cooperative usage. These dependencies are outside these schemas, in dependency schemas that define interdatabase relationships with respect to the interdatabase integrity, privacy, or data meanings. Different groups of administrators may define dependencies independently.
- Users have at their disposal the multidatabase manipulation language Multidatabase Data Sublanguage, or MDSL. Unlike the present manipulation languages, MDSL makes it possible to express retrievals and updates, addressing jointly data in distinct database schemas, as well as to exchange data between databases. The language design is particularly oriented toward the simplicity of a multidatadase query expression. "Simplicity" here means that the user intention becomes a single (formal) query (statement), despite eventual semantic conflicts between the participating schemas. At present, the manipulations the MRDSM user may perform are basically as follows:
  - the joining of data in different database schemas;
  - the broadcasting of user intentions over a number of database schemas with the same or different naming rules for data with similar meanings;
  - the broadcasting of user intentions over a number of databases with data similar in meaning, but with different decomposition into relations;
  - the dynamic transforming of actual attribute meanings, units of measure, etc., into user-defined value types;
  - allowing data to flow between databases (interdatabase queries); and
  - the dynamic aggregating of data from different databases using various new standard (built-in) functions.

Experience with the MRDSM system and everyday use of Teletel databases show that all the above possibilities, at both the data definition and manipulation levels, are highly desirable. Most of them are still unique to MRDSM.

Data definition level

Database schemas. Each database schema presented to MRDSM for multidatabase use is defined through the MRDS data definition language. A schema may be a conceptual schema of an MRDS database, its data model in MRDS terminology, or a database view schema, also called a data submodel. In the latter case, the administrator can hide some parts of the conceptual schema from MRDSM users. The submodel may in particular be secured.[10]

Following the terminology of Heimbigner and McLeod,[4] all such schemas are called export schemas. For the users, export schemas constitute the conceptual schemas of the databases. The users have at their disposal commands enabling them to instantly display the export schemas of databases they wish to acquire knowledge of.

Database access. A database is visible to MRDSM users only after its declaration to the system. The perception the users then have of the database is defined by the export schema. In particular, the access rights that any user can have to the database are then bound by those defined in the schema. These rights are defined by the administrator through standard MRDS facilities. The administrator retains total control over these rights and, in particular, may change them at any moment. He also may at any moment withdraw the database from MRDSM. The database then becomes invisible again to the MRDSM users.

Multidatabase naming. An administrator or a group of administrators using MRDSM may define for any collection of databases a collective name called a multidatabase name. For instance, the databases Michelin, Kleber, and Gault_M may collectively get the name Rest_guides. Collective names are popular with public database servers. Such names may also simplify the expression of some commands. Otherwise, these commands may require an enumeration of the corresponding databases. A multidatabase name may itself be an element of a larger collection provided with the multidatabase name, etc.

Database identification. Different users may choose the same database or multidatabase name for different databases. The corresponding MRDSM identification rules extend those of MRDS. The MRDSM rules are as follows:

- Any collection of databases managed by MRDSM is called a multidatabase. All databases and, possibly, named multidatabases of the same user implicitly constitute a multidatabase called user multidatabase, the name of which is the user's name. Then, all user multidatabases of the same Multics project implicitly constitute the project multidatabase named as the project is. Furthermore, all project multidatabases in the same MRDSM site constitute a site multidatabase, named INRIA for instance, etc.
- Names given to (multi)databases when they are created are called relative names. The user may refer to his own (multi)databases using only the relative names. To refer to a (multi)database of another user in the same project, the user may prefix the relative name with the corresponding user name. Another possibility is to move into the other user directory, in which case relative names suffice. Similar rules govern access to databases of other projects, etc.

Interdatabase dependencies. The administrators may at present define three types of dependencies:

- manipulation dependencies,
- privacy dependencies, and
- equivalence dependencies.

Manipulation dependencies. A manipulation dependency triggers a query to a database when a given query to another database occurs. For instance, an insertion of data about a new restaurant to one user database may trigger the insertion of the same data to his friend's restaurant database. Manipulation dependencies may thus act as a kind of message passing system. In particular, a triggered query may in turn become the source of another query, etc.

The triggered queries are called complements of the original query. To declare a complement one has to indicate (1) the relation name to be concerned by the source manipulation, (2) the type of the source query (insertion, deletion, etc.), and (3) the name of the Multics segment with the complement itself.

Manipulation dependencies may be defined independently. Through transitivities, a manipulation may then lead to complements outside the original administrator's knowledge. Long and even infinite chains of complements may then appear. MRDSM does not prevent such a situation. However, it limits to a predefined value the number of complements to be processed. The set of complements is determined before the query execution. If the limit, let it be N, is exceeded, a warning is issued. In any case, at most N complements are processed.

While defining the complement, the administrator may indicate whether its execution should precede or should follow the source manipulation. In the former case, the source manipulation is performed only if the complement execution could take place. In the latter case, the administrator may ask for a deferred execution.

Privacy dependencies. The aim of these dependencies is to prohibit manipulations that would match data from different databases in a way that would disclose confidential information. The declaration of such dependencies is similar to that for the manipulation dependencies. However, the privacy dependencies may access the user identification data. They may further correspond to arbitrary checks whose detailed principles are outside the MRDSM design goals. Their basic action is nevertheless to include selections and projections with every query posed by a designated user about a designated relation. These selections and projections have the effect of making certain values invisible.

Equivalence dependencies. These dependencies identify in different databases the primary or candidate keys whose equality of values corresponds to the same real object. For instance, such a dependency could be declared for the restaurant names and corresponding street names in the case of our example databases. They are useful for formulation of some multidatabase queries with implicit joins. We will present these types of queries in the next section.

MRDSM data manipulation language

An overview. The MRDSM data manipulation language called MDSL extends DSL,[10] the data manipulation language, or DML, of MRDS. Like SQL and QUEL, DSL is based on tuple calculus. New functions within MDSL are mainly intended for multidatabase queries. Some auxiliary functions are designed for query editing, help, and displaying of schemas. Also, any Multics command may be called. Thus, text editors may be called, programs may be compiled or executed, etc. The main functions are designed to make it possible to formulate multidatabase queries of the types discussed earlier. Only these functions will be presented in the following discussion.

Query form. The general form of a MDSL query 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 (r for retrieve, u for update in shared mode, and either er or eu [default mode] in exclusive mode). 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 a query refers to are called the 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 and will be presented below. The clauses -range, -select, and -where are main clauses. Their syntax is basically similar to those of DSL. The semantics will be presented below. The -value clause exists only for updates. The 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.[12] 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, etc.). 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 reasons will be shown later on.

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 relation, or an update to a database relation. DSL queries, as well as queries formulated using known relational DBSs, are all elementary monodatabase queries. An elementary multidatabase query differs from a DSL query in that designators may concern relations in different schemas. The -where clause of any elementary multidatabase query then involves interdatabase joins. The corresponding implementation issues in MRDSM are presented in Litwin.[8]

Unlike a DSL query, an elementary MDSL query makes it possible to use 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 on the same street.

```
open My_rest Cinemas er
-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. The mode er is valid for both databases.

Multiple queries

The concept. Multiple queries are intended for situations where various data-

bases model the same universe. The user may then need to broadcast the same manipulation to several databases (e.g., project any relation describing restaurants on the attribute expressing the restaurant type). Present relational languages do not allow one to simply express such intentions. If only elementary queries are available, then the user needs to formulate as many queries as there are databases. Furthermore, these queries may differ from database to database. In contrast, multiple queries allow one to broadcast the intention through a single query. This may be a considerable simplification, especially for larger scopes. Multiple query has also been called diffusion query or broadcast query.[7]

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 a set of all elementary queries, called subqueries, that may result from all choices of unique identifiers within the designator scopes. A choice may lead to a subquery that cannot be executed. We call the executable subqueries pertinent. 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 new concepts of (1) multiple identifiers, (2) semantic variables, and (3) options on the target list.[12]

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 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 the following 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 the relations. The location would then be

Example databases

The schemas that follow define the databases we use throughout the examples. The relations are defined according to the MRDS data definition language. We avoided the domain and attribute declarations, which are 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 of 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 those of the actual guides and due to the databases' autonomy:

1. Guides partly disagree upon (a) the choice of attributes that should model the universe of restaurants and (b) 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, where m = 1,...,20. There is no objective specific rule for ratings correspondence. Nevertheless, it is frequently clear that guides disagree about a restaurant.
5. The guides may also disagree on the average price of a meal or on a restaurant type. For instance, a restaurant may be Chinese for one guide and Vietnamese for another. The guides may disagree also on the phone number, although it is a candidate key within each database.
6. In particular, Michelin and Gault__M disagree on the meaning of attributes dealing with prices, despite the same attribute names. Michelin prices are without the 15 percent tip, mandatory in France, while Gault__M prices include the tip.
7. In contrast, the guides always agree on a restaurant name and the corresponding street name. This property thus identifies the same restaurant in different guides. Likewise, course (dish) names are the same in different guides.

Similar properties will be typical of the general multidatabase environment. Multiple databases relative to the same universe will usually resemble each other, but will also present numerous semantic differences like those above.

semantically meaningful, as it would implicitly indicate the guide that considers the restaurant 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 in Michelin that have attribute r. It would thus replace two classical relational queries. In addition, it would automatically preserve the referential integrity. This is not the case with queries one may formulate using known relational languages.[13]

Semantic variables. We call a 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, or implicit, which means that they result from the variable name. The aim in the concept 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 ($x_1.x_2. \ldots .x_k\ n_{1,1}.n_{2,1}. \ldots .n_{k,1}\ n_{1,2}.n_{2,2}. \ldots n_{k,2}\ldots)\ldots$

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

Example 4. (Q2) Retrieve from Rest_guides, i.e. from Michelin, Kleber, and Gault_M databases, restaurants that a guide considers to be 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 francs.

```
open Cinemas r My_rest r
-db (cc Cinemas)(mr My_rest)
-range_s (x.y.y# R.M.r# C.P.c#)
-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, and ?, designating any but only one digit. The domain is then implicitly constituted from all data names in the scope (not data values, like in SQL) that match the resulting pattern. For instance, the domain of x = R* is all names in the scope that start with R. The subqueries correspond to all pertinent substitutions of data names within the domains. If the characters * and ? are parts of data names themselves, as in R* for instance, then they should be preceded by the character \. Thus x = R* would include all names starting with the string R*.

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

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

This formulation would remain valid for any number of databases in the scope, provided that all and only restaurant relation names start with the character R. The corresponding query with the explicit domain would then be usually more complex, as it would require a range_s clause with as many identifiers as there are different names.

Options. Current relational DMLs assume implicitly that all the attributes in the -select clause target list are in the schema of the addressed database. As the examples below will show, it is useful to relax this assumption in the multidatabase environment. The concept of options[12] is intended for this purpose. The corresponding syntax is as follows.

Let d be an attribute designator within the -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 a space, as is usual in DSL, then q is pertinent only if there is 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 $d_1|d_2|\ldots|d_n$ 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 were mandatory, the tuples would be retrieved only from the 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.

Incomplete queries

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.[14] They are deduced by the system from database schemas. The result is called a complete query. The completion algorithm is described in detail in Litwin.[14]

It is shown that this process leads to the intuitively expected result in more cases than the present algorithms for the universal relation interface.[13] A major consequence is also that updates may be performed.

This function has as goals (1) to further simplify query formulation and (2) to allow multiple queries to databases modeling the same universe, but different through decompositions into relations. Indeed, there is 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                                (Q3)
-select street
-where (cname = "confit d'oie")
retrieve
```

The complete query would be

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

Example 12. Delete from My_rest all the courses whose ncal > 2,000.

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

This will delete the appropriate tuples from both C and M.

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.[9] This function makes it possible for the user to dynamically and subjectively transform data values. Such a need will be frequent in the multidatabase environment. In particular, it will be frequently necessary for interdatabase joins, as joins are meaningful only for data with 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
-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) = (a1, s1),...,(ak, sk)
-define by F(s) = arithmetical_formula
-define by P(s) =
    Multics_segment_name
```

s_i are actual values and a_i 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 there are any. This clause is currently mandatory if MT in the -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_i such that $a' = a_i$. 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 then designates 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 the hold argument. Then, the -select clause has to be preceded by the clause -actual n.

Example 13. Assume that the '****' rating of Michelin corresponds to Gault_M.qual ≥ 19/20 and '***' corresponds to 17/20 ≤ qual ≤ 18/20. 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 the 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 dynamically computes through the Multics execute command the values of stars. It expresses, in an arbitrary host language, the algorithm:

```
if qual ≥19/20: stars = '****' else if
   qual ≥17/20: stars = '***' endif;
```

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

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

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 here renders meaningful the interdatabase join on price, 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 Litwin and Vigier.[9]

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. 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 onto target attributes with the same names by default, unless specified otherwise within the matching list. The rule "by order" means in contrast that the attributes should be matched in order of their listing in the source -select clause, to the attributes in the matching list if one is present, or those of the target schema; otherwise, in the prescribed order. In the former case, the elements in the matching list must be target names only.

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 'new' are disregarded. Conversely, 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, the 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's opinion about a restaurant, a course, or a menu, will have priority. The inverse effect would appear if the modify command were used.

This query represents the case we spoke about in the overview of MRDSM, where source data in several relations should enter several target relations. Some other instances where such a case would arise are a supplier and his parts, a student and 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 the '***' 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
```

Standard functions. Standard functions, such as avg, sum, max, etc., may be declared in MDSL in two ways:

(1) Inside the clauses, being then enclosed within square brackets. The function is then evaluated independently for each subquery.

(2) As independent clauses. The function applies then to all the tuples of the query.

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

Example 19. Retrieve the average price of meals according to each database in Rest_guides.

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

This query returns three values, one for each database. The following returns just one value.

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

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

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. Then name(n) provides the name of data designated by n, name(.n) provides the name of the container of data (relation for an attribute, etc.) designated by n, name(..n) refers to 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 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. The norm function has the following syntax:

-norm ((id_att) tuple_var).

Here 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 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 a 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. Discussion of such algorithms may be found in Dayal.[15]

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 10 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 $n \geq 1$ tuples sharing the values of attributes designated in the list A. Priorities corre-

spond 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 Chinese restaurants from Rest_guides, but limit the number of descriptions of each restaurant 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.

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 present languages were designed for manipulations of single databases. These functions should especially allow users to manipulate autonomous databases. MRDSM offers some such functions for relational databases. Most future distributed databases will be of this type or will at least present a relational view. Numerous examples show that the MRDSM functions should prove useful for a large variety of user needs. This is because of their flexibility and open nature with respect to the accommodation of autonomous names, values, and structures. Most of these functions are not yet available in other languages and systems. The functions designed for MRDS databases can also be added to other relational languages. The corresponding extension of SQL is under study. The concepts of multiple identifiers, semantic variables, interdatabase queries, etc., may further be applied to other data models. They may thus form a useful basis for the design of distributed systems using other popular models.

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 in any relational system.

The functions discussed lead to many open problems at the implementation level. Also, new functions may be added. A function currently studied for implementation concerns multidatabase views, to be defined through stored MDSL queries, in the manner similar to those of DB2 and of INGRES. Knowledge processing techniques are also investigated, as they should prove particularly useful in the multidatabase environment.[2,9] In particular, they should enlarge the class of intentions expressible as a single query and should make it possible to further simplify the expression of some queries.□

References

1. D. Heimbigner and D. McLeod, "A Federated Architecture for Information Management," *ACM-TOIS*, Vol. 3, N3, 1985, pp. 253-278.
2. C. Hewitt and P. De Jong, "Open Systems," *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, eds. M. L. Brodie, J. Mylopoulos, and J. W. Schmidt, Springer-Verlag, Berlin, 1985, pp.147-164.
3. P. Selinger et al., *The Impact of Site Autonomy on R*. Databases-Role and Structure*, Cambridge Univ. Press, New York, 1984, pp. 151-176.
4. S. Ceri and G. Pelagatti, *Distributed Databases Principles & Systems*, McGraw-Hill, New York, 1984.
5. W. Litwin et al., "SIRIUS Systems for Distributed Data Management," *Distributed Databases*, ed. H. Schneider, North-Holland, New York, 1982, pp. 311-366.
6. T. Landers and R. L. Rosenberg, "An Overview of MULTIBASE," *Second Symp. Distr. Databases*,. Berlin, Sept. 1982, North-Holland, New York, 1982, pp. 153-184.
7. P. Lyngback and D. McLeod, "An Approach to Object Sharing in Distributed Database Systems," *VLDB 83*, Florence, Oct. 1983, pp. 364-376.
8. W. Litwin, "An Overview of the Multidatabase System MRDSM," *ACM Nat'l Conf.*, Denver, Oct. 1985.
9. W. Litwin and Ph. Vigier, "Dynamic Attributes in the Multidatabase System MRDSM," *IEEE-COMPDEC-2*, Los Angeles, Feb. 1986.
10. *Multics Relational Data Store (MRDS) Reference Manual*, CII-Honeywell bull. ref. 68 A2 AW53 REV4, Jan. 1982.
11. *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, eds. M. L. Brodie, J. Mylopoulos, and J. W. Schmidt, Springer-Verlag, New York, 1984.
12. W. Litwin, "Concepts for Multidatabase Manipulation Languages," *JCIT-4*, Jerusalem, June 1984, pp. 433 and 442.
13. J. D. Ullman, *Principles of Database Systems*, second ed., Computer Science Press, Rockville, Md., 1983.
14. W. Litwin, "Implicit Joins in the Multidatabase System MRDSM," *IEEE-COMPSAC*, Chicago, Oct. 1985.
15. U. Dayal, "Query Processing in a Multidatabase System," *Query Processing; Database Systems*, ed. Kime et al., Springer-Verlag, New York, 1985, pp. 81-108.

Witold Litwin has been research director at the Institut National de Récherche en Informatique et Automatique, or INRIA, since 1977. He is mainly interested in the design of multidatabase management and distributed systems and in dynamic hashing data structures. He currently leads the SESAME project at INRIA. He is also the author of several papers, the editor of three books, and a referee for ACM, IEEE, and NSF.

Litwin received his PhD in Computer Science in 1971 and Ès-Science doctor degree in mathematics from the University of Paris 6 in 1979.

Readers may write to Litwin at INRIA, BP 106, 78153 Le Chesnay Cedex, France.

Abdelaziz Abdellatif is a doctoral candidate in computer science at the University of Pierre et Marie Curie, Paris. Since 1983 he has been participating in the design of the multidatabase system MRDSM at SESAME Project at the Institut National de Récherche en Informatique et Automatique.

Abdellatif received the Diplome Universitaire des Études Scientifiques in 1979 and the engineering degree with a computer science specialization in 1983 from the University of Tunis. In 1984, he received the Diplome des Études Approfondies in computer systems from the University of Pierre et Marie Curie.

Readers may write to Abdellatif at SESAME Project, INRIA, BP 105, 78153 Le Chesnay Cedex, France.