INFORMATION SCIENCES **49**, 59–101 (1989)

MSQL: A Multidatabase Language

W. LITWIN
A. ABDELLATIF
A. ZEROUAL
B. NICOLAS

and

PH. VIGIER

INRIA, BP 105, 78153 Le-Chesnay, France

Communicated by Ahmed Elmagarmid

ABSTRACT

Many databases are now accessible through computer networks, and users frequently have data of interest in different databases. Manipulations of such data call for functions that do not exist in classical manipulation languages and database systems. A new type of data manipulation languages is needed, called multidatabase manipulation languages. We describe the multidatabase language MSQL, designed for the manipulation of data from different relational databases. MSQL is an extension of SQL, which has become the standard for relational database systems.

1. INTRODUCTION

Many databases now exist on various computers and are accessible through computer networks. A large mainframe with a database system like INGRES or a database server like Compuservice may support dozens of databases. A videotex system like Teletel may provide access to thousands of databases for millions of people. A major consequence is that many users now have data of interest in different databases. Such users frequently need to jointly manipulate data in different databases. A Teletel user who wishes to go out to a movie and for dinner may wish to find, in a cinema guide database and in a restaurant guide database, the cinemas and restaurants that are in the same street. A bank may have several branch databases, and the manager of a company may wish the balances of the accounts that the company has at different branches. If this

655 Avenue of the Americas, New York, NY 10010 0020-0255/89/$03.50

manager uses Teletel, he may further wish to have the possibility of making such a query to all the banks his enterprise uses (most French banks are accessible through Teletel). Then, a user of Teletel who wants to find the cheapest way to travel somewhere may need to query several airline databases and maybe the SNCF (French railways) database. Furthermore, if a street changes name, it may be necessary to update all the corresponding databases, etc.

Manipulations of such data require functions that do not exist in classical manipulation languages and database systems [17], being designed for manipulations of data integrated into one database. One reason is that data in different databases are not usually mutually integrated, as, for various reasons, the administrators wish to remain independent. Data about the same real things (accounts, restaurants, etc.) then present semantic differences. The differences consist of partial redundancy; differences in naming, data structures, types, and scales; independence of keys; etc. This type of heterogeneity is independent of a data model, which means that it exists even if all databases present the common data model and manipulation language for cooperative usage. The differences among similar data result from different perceptions of the same real universe, from local needs, from the competition for customers (e.g. restaurant guides), from political incompatibility (French Socialist Party database and French National Front database), etc. Usually, in such a case, there is no way to force integration through a global schema, since no one would agree on the global administrator. The databases shown in the appendix, modeling actual databases, show some of the differences that may occur.

A system for the manipulation of data in autonomous databases is called a *multidatabase system* (MBS) [14]. By the same token, the corresponding manipulation language is called a *multidatabase manipulation language* (MML). Databases that may be manipulated together without global integration are called *interoperable* [17]. Relational databases are a particularly important subject for interoperability, since most future databases will present a relational interface, no matter which data model and implementation issues are used internally. [17] describes the prototype relational MBS termed MRDSM and its MML, termed MDSL. MRDSM makes interoperable databases managed by the MRDS system that is the relational DBMS of the Multics system (Multics Relational Data Store [22]). Most of the new functions of MDSL are unknown in other languages.

MDSL is an extension of the MRDS manipulation language DSL. This language is particular to the MRDS system, although its syntax is quite typical of relational languages, being somewhere between QUEL and SQL. Nevertheless, it is clear today that the future standard language for a relational database will be SQL. This language is therefore at present the best basis for multidatabase interoperability.

Below, we present the multidatabase extension of SQL, termed MSQL (multidatabase SQL) intended for this usage. Any function of SQL is by definition the function of MSQL. New functions are designed for nonprocedural manipulation of data in different and basically mutually nonintegrated SQL databases. This means that the user's wish (informal query) should usually become a single MSQL statement. The overall design of MSQL is based on that of MDSL, with adaptations to SQL syntax and semantics. The definition of SQL used is mainly that of ISO [25] and of the DB2 dialect [6]. We assume both of these definitions and the main properties of SQL that result from them to be known. The new possibilities that MSQL statements provide for interoperability are as follows:

(1) single statement table creation or alteration in any number of databases and the import of the database, table, or column schemas;

(2) retrieval or modification involving the joining of data in different database schemas;

(3) the broadcasting of a retrieval or of a modification over any number of databases where data with similar meanings have the same or different naming rules or decompositions into relations or value types;

(4) the dynamic transformation of actual attribute meanings, units of measure, etc., into user-defined value types that may be retrieved or updated (some restrictions may apply to updates);

(5) interdatabase queries for data flow between databases;

(6) the dynamic aggregation of data from different databases using various new standard (built-in) functions;

(7) the creation of multidatabase views and of virtual databases over such views;

(8) the creation of auxiliary objects like triggers, stored queries, and transactions (procedures).

Section 2 presents MSQL data definition statements. Section 3 deals with functions for multidatabase retrieval and update. Section 4 presents the views. Interdatabase queries are discussed in Section 5, and auxiliary objects in Section 6. Section 7 concludes the MSQL description.

2. DATA DEFINITION

2.1. INTRODUCTION

As in [2], in what follows the term *database* represents a (logical) model of a certain company (also called universe, real universe, real world, or world in some implementations of Prolog [21]. It should not be confused with a meaningless physical storage structure called a (physical) database in some imple-

mentations (DB2 in particular [6]), or with the physical concept of a site (network node). The databases are usually meaningfully named after their universes, like database **AIR FRANCE** and the databases in the Appendix, and unlike "physical" databases and sites (e.g., the site name **GCAM** of the **AIR FRANCE** database is meaningless for database users). Finally, the databases are defined by their schemas and data models, no matter how they are implemented internally. For instance, a database implemented using IMS, but where the user perceives data as tables manipulated using SQL, is a relational database (note: this is in particular the approach in [6]). Below, we deal with relational and SQL databases only.

The relational database design follows the idea of integration. The basic data structure for the integration is a table that is a set of named columns whose values are atomic and of the same value type. Data about all (real) objects (entities or relationships) of the same type are supposed to be integrated into the same table. The objects of the type Suppliers are, for instance, supposed to constitute a single table, usually named S. The drawback is that each object has to be modeled basically through exactly the same column names and value types, unless null values are used with all the problems they create. The advantage is the simplicity of relational languages for interactive usage that is otherwise lost.

EXAMPLE 2.1. If all suppliers are described by the same table S, then the wish to select all data about all suppliers leads to the simplest possible SQL query:

```
SELECT *
FROM S
```

If however the suppliers were in $i > 1$ tables, for instance one per department the suppliers deal with, then the expression of the same wish will need i queries like the above one in any relational database language. If i is large, the usage of SQL becomes laborious, since it is highly procedural; if i is unknown, then the wish cannot be expressed at all.

The queries using column names may in addition differ in formulation if table schemas are particularized. Aggregate operations like **GROUP BY** or **SUM** may moreover be impossible to express if the tuples to aggregate are in different tables. While the departments of Example 2.1 may therefore find it necessary to be integrated into a single table, they may remain unhappy with the compromises they had to adopt to get the common schema. This is in fact the situation for many real applications.

In a multidatabase environment the integration principle is naturally applied up to the level of a database. Afterwards, as data are in several databases that are naturally autonomous, objects of the same type may be modeled by several

tables. These tables may bear the same or different names and may have the same or different columns. If the users use the same natural languages and the administrators work with some cooperation, as is usual in an enterprise or even when organizations compete for customers, then the usual situation is as follows:

(1) Some tables and columns designate the same things using the same name and value types (e.g., price is called **price** in all tables and its value type is FFr). This uniqueness may result from cooperative standards adopted by the autonomous administrators.

(2) In contrast, some tables have particular names and/or bear columns particular to the database. These columns may have a particular name (ex. **prices**) or a particular value type (ex. price with VAT included) or may even have no semantic equivalent in other databases (no notion of price at all).

(3) Some real objects may be modeled by only one table, while some other may correspond to tuples in several tables (e.g. a supplier working for several departments). The values may be duplicated, or may differ in precision, or may have no objective (univocal) relationship, or may be clearly contradictory.

This "right to be different" seems highly appreciated by users and appears to be the motivation behind the current trend towards multiple autonomous databases, instead of a single integrated one (more important than considerations of physical performance). Though different tables may then be sometimes presented through a view as a single table, this approach does not seem the natural issue (either conceptually or technically, because of update problems). The natural goal seems rather to be *an extension of relational principles and statements to sets of tables* that may appear subject to such needs in the multidatabase environment. The set of tables *implicitly or explicitly* subject to a common statement will be called *multitable*. In Example 2.1, the multitable would consist of all the supplier tables of all departments and could of course span over multiple databases. In particular, the application of the relational principles implies that a manipulation of a multitable should be expressible through a single statement, at least whenever the same statement would apply if data in a multitable were integrated into a table. In this sense the manipulation of multitables would become as simple as that of tables. The tables of a multitable become interoperable, providing the user with both relational simplicity and better recognition of his autonomy.

MSQL is designed particularly with this goal in mind, and it was also implicitly the goal of MDSL [17]. SQL capabilities are extended to multitables whenever it appears useful to do so. In particular, they make it possible to create multitables at data definition level and to form multitables corresponding to various wishes dynamically in queries. The corresponding functions and constructs of MSQL will now be progressively introduced.

2.2 STATEMENTS

As in SQL, the basic construct for data manipulation and definition in MSQL is a *statement*. By definition, any SQL statement is an MSQL statement. In what follows, we present only the statements and clauses relative to the logical level of data. The details relative to storage space, physical clustering, and network access are assumed system-specific.

The data definition statements of MSQL that have been extended with respect to SQL are as follows:

CREATE TABLE	**CREATE DATABASE**
CREATE MULTIDATABASE	**CREATE VIEW**
ALTER TABLE	**ALTER VIEW**
ALTER MULTIDATABASE	
DROP TABLE	**DROP DATABASE**
DROP MULTIDATABASE	**DROP VIEW**

We defer the discussion of the statements relative to views to Section 5, as they require the knowledge of MSQL data manipulation statements. Some statements for the definition of auxiliary database objects will also be shown in Section 6.

2.3. NAMING RULES

SQL makes it possible to name two data types: *columns* (attributes)—also called fields—and *tables*. MSQL allows us to name *domains*, *databases*, and *multidatabases*. To refer to data or variable names, statements contain *designators*. Data designators in SQL are column, table, or view identifiers. They are unique (unambiguous) identifiers that are either column, table, or view names, or unambiguous aliases, or qualified column names that consist of a table name and the column name separated by a period. In addition, MSQL provides the following types of designators:

(1) the database names,
(2) the multidatabase names,
(3) virtual database names,
(4) multiple identifiers,
(5) domain names,
(6) semantic variables,
(7) dynamic column names.

The last two types of designators are used only in data manipulation statements and will be discussed in the corresponding sections. The database name is the name given to a relational database. The multidatabase name is the name given to a set of interoperable relational databases. A virtual database name is used to designate a set of tables or views of one or several databases. A multiple identifier is the name shared by several objects in a statement scope, especially tables in different databases or columns in different tables. The usage of all these types of designators will be explained during the discussion of the details of the statements.

The notion of domain, though fundamental to the relational model, does not exist in SQL, in contrast to DSL (see for instance [22]). This is regrettable from both a theoretical and a practical point of view [9], in particular because the existence of domains can usually remove the need for equijoins [15]. As will be shown later and is also shown in detail in [15], this possibility is even more important in the multidatabase environment, as it allows the query formulation to be largely independent of data structure heterogeneity (differences in decompositions to relations). That is why, unlike SQL, MSQL supports domains.

One way to define domains is that called the extended SQL in [9]. Another way is to consider that any column name is either a domain name, or a qualified domain name if it is of the form ⟨column prefix⟩_⟨domain name⟩, as in [4]. The latter approach requires more discipline while defining the columns, as each column definition over the same domain should contain the same data type. However, it is compatible with current SQL implementations. In what follows we retain the second approach, although the corresponding features of MSQL would remain for the former approach as well.

2.4. DATABASE CREATION

The following statement creates the database, either empty or with tables or views and the corresponding privileges:

CREATE DATABASE ⟨database name⟩
[⟨table definition⟩|⟨view definition⟩|⟨privilege definition⟩...]|
FROM ⟨database identifier⟩

⟨table definition⟩ :: = see Section 2.8
⟨view definition⟩ :: = see Section 4.1
⟨privilege definition⟩ :: = see Section 2.5

This statement must precede the creation of any table. As usual, square brackets in syntactic definitions of this and coming statements indicate that the

material enclosed is optional. A "|" indicates "or." Ellipses "..." indicate that the material enclosed may be repeated zero or more times. The form {⟨a⟩|⟨b⟩} means that one of the elements is mandatory. Finally, material in ⟨ ⟩ must be replaced by specific values chosen by the user.

The semantics of the statement includes that of the SQL statement; new features with respect to table or view definitions will be shown in the corresponding sections. The **FROM** clause is new with respect to SQL and results from the multidatabase environment. It makes it possible to import (copy) the entire schema of an existing database.

EXAMPLE 2.2. Create a new branch database **boulogne** upon the database **bnp**.

CREATE boulogne FROM bnp

The SQL statement would be much more procedural and subject to more errors, since it would require the whole schema of **bnp** to be retyped.

2.5. *PRIVILEGE DEFINITION*

The following statement defines the user's privileges or imports them. It may be used in the **CREATE DATABASE** statement or separately:

GRANT⟨privileges⟩[**ON** [⟨object type⟩]⟨object designators⟩] **TO** ⟨grantee⟩...
[**WITH GRANT OPTION**] [**FROM** [⟨grantee⟩ **ON**]⟨object type⟩⟨object identifier⟩]

⟨privileges⟩ :: = **ALL [PRIVILEGES]**|⟨action⟩[,⟨action⟩...]
⟨action⟩ :: = ⟨MSQL statement⟩[(⟨grant column list⟩)]
⟨grant column list⟩ :: = ⟨column designator⟩[,⟨column designator⟩...]
⟨grantee⟩ :: = **PUBLIC**|⟨authorization identifier⟩

The statement semantic is basically that of the SQL **GRANT** statement; see for instance [6] for a deeper discussion. The difference is that MSQL has more statements and object types. In particular the statement may concern one or more multitables of tables sharing the name, if it is applied independently of the **CREATE DATABASE** statement. The corresponding table designator(s) should then be multiple identifier(s). Furthermore, the statement may import all privileges the grantee(s) had on the object identified in the **FROM** clause. The object identifier in the clause should then be a unique identifier, qualified if

necessary. Also, the object type in this clause and those concerned by the statement should be the same.

EXAMPLE 2.3. The following statements authorize Nicolas and Abdellatif to select data from any of four **client** tables in the databases in the Appendix and import to the **account** table in **etoile** database their respective and possibly different privileges on the **account** table in **bnp**. The last statement imports for Nicolas the privileges of Zeroual:

USE bnp sg cic etoile
GRANT SELECT ON client TO Nicolas Abdellatif
GRANT ALL ON etoile.account TO Nicolas Abdellatif FROM bnp.account
GRANT ALL ON etoile.account TO Nicolas FROM Zeroual ON bnp.account

The **USE** clause is optional and discussed in Section 3.2. The designator **client** is the multiple identifier that defines dynamically the corresponding multitable. The corresponding statement replaces four SQL **GRANT** statements. The utility of the other two statements is obvious.

2.6. *PRIVILEGE REVOKING*

Revoking privileges is done by the following statement, also similar to that of SQL:

REVOKE⟨privileges⟩**ON**⟨designator⟩**FROM**⟨grantee⟩...

2.7. *DATABASE REMOVAL*

The following statement removes the designated database:

DROP DATABASE⟨database name⟩

It is particularly useful for temporary databases, such as a frequently refreshed database on a PC containing part of mainframe database.

2.8. *TABLE CREATION*

The SQL statement **CREATE TABLE** creates a table schema in the database that is known implicitly. In a multidatabase environment, it may be necessary

to indicate the database concerned. Furthermore, one may need to create several tables in cooperating databases simultaneously. Finally, it may be useful to import schemas of existing tables or columns. The MSQL **CREATE TABLE** statement includes the SQL statement and satisfies the new needs:

[USE⟨scope specifications⟩**]**
CREATE TABLE⟨table designator⟩(⟨column definition⟩[,⟨column definition⟩]...)|
FROM⟨source table identifier⟩

⟨column definition⟩ :: = ⟨column name⟩⟨data type⟩**[NON NULL [UNIQUE]]|**
CONSTRAINTS⟨unique constraint definition⟩...|
FROM⟨source column identifier⟩

⟨unique constraint definition⟩ :: = as in [25]

The **USE** clause designates the names of the (multi)databases the statement applies to (the scope of the statement). It may be omitted if the current scope is used. The **FROM** clause identifies the table or the column whose schema is imported. The statement creates the tables named upon the ⟨table designator⟩ in all databases in the considered scope. As in SQL [25], the **UNIQUE** statement defines the keys.

EXAMPLE 2.4. The example banks have decided to cooperate on loans. They have nominated a committee for the basic definition of the corresponding table, with the **CREATE TABLE** privilege on all databases. The committee has decided that each loan will be a tuple in the following table in each database:

loan (acc#, cl#, amount, rate, beg_date, duration)

The following **CREATE TABLE** could be used:

CREATE TABLE loan (acc# INT NON NULL, cl# INT NON NULL,
CONSTRAINTS UNIQUE (acc#, cl#),
amount INT,
rate INT,
beg_date FROM bnp.open_date,
duration INT)

The common format and constraints adopted for **beg_date** are the same as those of **open_date** in **bnp**. The statement replaces as many SQL statements as there are databases in the current scope. If afterwards **loan** should be imported to a new database, such as **boulogne**, the following statement would do it:

CREATE TABLE boulogne.loan FROM bnp.loan

2.9. *TABLE EXPANSION*

MSQL extends the SQL **ALTER TABLE** statement as follows:

[USE⟨scope specifications⟩**]**
ALTER TABLE⟨table designator⟩ [⟨table designator⟩...]
ADD⟨column designator⟩{⟨data type⟩|**FROM**⟨column identifier⟩}
MODIFY⟨column designator⟩{⟨data type⟩|**FROM**⟨column identifier⟩}

The statement adds a column to all the tables designated in the databases that are in the scope or modify the data type of a column. The column schema may be imported. Designators may be unique or the multiple identifiers. They may also contain the SQL generic characters "_", "%", or "*", designating then all tables in the scope whose names conform to the resulting pattern. This possibility, as well as the new possibility of having the list of table designators, is provided for very frequent cases where similar tables are to some extent named differently.

EXAMPLE 2.5. The example banks have decided to add a **manager_name** column to their tables describing the branch offices. They may use the following single statement:

USE bnp cic sg
ALTER TABLE br% ADD manager_name CHAR(80)

The statement is equivalent to three SQL **ALTER TABLE** statements. An alternative statement for the same scope could be

ALTER TABLE br branch ADD manager_name CHAR(80)

To subsequently import the column to **etoile**, assuming **br** created, one could use the statement

ALTER TABLE etoile.br ADD manager_name FROM bnp.br

2.10. *TABLE REMOVAL*

The MSQL **DROP TABLE** statement is as follows:

[USE⟨scope specifications⟩**]**
DROP TABLE⟨table designator⟩ [⟨table designator⟩...]

As previously, unlike in SQL, one may use multiple identifiers and generic characters for table designation (the possibility of having the list is new as well). One may thus simultaneously destroy several tables with the same or different names.

2.11. *MULTIDATABASE CREATION*

A multidatabase is a set of interoperable databases or multidatabases. Some databases may be virtual, in the sense that their conceptual schemas are external schemas in the ANSI/SPARC sense. To declare a multidatabase means to name the set. The *multidatabase name* may then be used as the qualifier or as the single designator of the query scope. The declaration of a multidatabase is optional, but may obviously be useful. It may name databases of some federation (cooperating banks, airlines etc.). It may also designate some or all databases or multidatabases of a server etc.

The multidatabase creation statement has the following syntax:

CREATE MULTIDATABASE⟨multidatabase name⟩
(⟨(multi)database identifier⟩[, ⟨(multi)database identifier⟩]...)
⟨(multi)database identifier⟩ :: = ⟨database identifier⟩|⟨multidatabase identifier⟩
⟨database identifier⟩ :: = ⟨actual database identifier⟩|⟨virtual database identifier⟩

Identifiers are unique identifiers. Databases have to exist already, in contrast with the **CREATE DATABASE** statement. A database or a multidatabase may belong to several multidatabases. As a table name, a (multi)database name may be qualified with the name of the multidatabase it belongs to. Different databases may indeed share a name, as already sometimes happens for databases of different servers.

EXAMPLE 2.6. The bank databases may be declared as the multidatabase **Banks**:

CREATE MULTIDATABASE Banks (bnp cic sg)

The order has no importance.

2.12. *MULTIDATABASE MODIFICATION*

MSQL makes it possible to modify the multidatabase schemas dynamically through the following statement:

ALTER MULTIDATABASE ⟨multidatabase identifier⟩
INCLUDE ⟨(multi)database identifier⟩[, ⟨(multi)database identifier⟩]... |
REMOVE ⟨(multi)database identifier⟩[, ⟨(multi)database identifier⟩]...

The "or" is not exclusive, i.e., one may include and remove databases in the same **ALTER** statement. The database to be included must exist already. The removed database is not affected as concerns its data or schema. It remains interoperable with databases in the multidatabase, but is no longer in the scope of the multidatabase name.

EXAMPLE 2.7. The bank **Vernes** wishes to join the **Banks** multidatabase with its database (or maybe its multidatabase), while **cic** wishes to leave it:

ALTER Banks
INCLUDE Vernes
REMOVE cic

The next **USE Banks** statement will open **Vernes**, but not **cic**.

3 DATA RETRIEVAL AND UPDATE

3.1. *STATEMENTS*

The general form of an MSQL data manipulation statement is as follows:

⟨**USE** statement⟩
⟨auxiliary statement⟩
⟨**SELECT** statement⟩|⟨**INSERT** statement⟩|⟨**DELETE** statement⟩|⟨**UPDATE** statement⟩|
⟨**REPLACE** statement⟩|⟨**STORE** statement⟩|⟨**COPY** statement⟩

The ⟨**USE** statement⟩ opens (multi)databases, prior to any manipulation except the creation. It defines the scope of the query. The auxiliary statements are introduced specifically for the multidatabase environment. The statements **SELECT**,...,**COPY** are main statements.

3.2. USE *STATEMENT*

We define *queries* as requests for retrievals or updates. SQL is designed for queries to a default database. In contrast, MSQL is designed for manipulations of tables that may be explicitly in different databases. The SQL query scope is implicitly the default database. The MSQL query scope may be several databases. The **USE** statement defines this scope through the (multi)databases to be opened. In addition this statement makes it possible to define the aliases for database names, simplifying the qualification of table names:

USE [**CURRENT**]⟨(multi)database name⟩|(⟨(multi)database name⟩⟨alias⟩)[⟨(multi)database name⟩|(⟨(multi)database name⟩⟨alias⟩)]...

⟨alias⟩ :: = ⟨character string⟩

A multidatabase name concerns all the corresponding databases. The existing scope is disregarded, unless the keyword **CURRENT** is used. In this case, the scope is only extended with the (multi)databases in this statement.

EXAMPLE 3.1. The following statements open the databases **bnp, cic, sg,** then add **etoile** with **e** as alias, then restrict the **scope** to **cic**.

```
USE Banks
...
USE CURRENT (etoile e)
...
USE cic
```

If the multidatabase **Banks** was not declared, one should enumerate the databases.

3.3. *ELEMENTARY QUERIES*

An MSQL query is an *elementary query* if all the 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. SQL queries, like that in Example 2.1, as well as queries formulated using known relational DBSs, are all *elementary monodatabase queries*. An *elementary multidatabase query* differs only in that it designates relations in different schemas. The **WHERE** statement of any elementary multidatabase query then involves interdatabase joins (if no implicit join introduced in Section 3.10 is allowed).

In the multidatabase environment, it may in particular occur that relations in different databases bear the same name. The basic new feature of an elementary MSQL query is that one may then use database names as prefixes for unique identification of such relations. Also, the **FROM** clause may be omitted if no ambiguity results from it. Other new possibilities will be shown progressively.

EXAMPLE 3.2. Retrieve all branches of **bnp** and **sg** that are in the same street as another branch:

```
USE Banks
SELECT brname, braname, street
FROM bnp.br, branch
WHERE bnp.br.street = branch.street
```

The table **br** is qualified to distinguish it from **cic.br**.

EXAMPLE 3.3. Add a 500 FFr bonus to clients in **bnp** who keep more money in this bank than in **sg**:

```
USE (bnp b) sg
UPDATE account
SET account.balance = account.balance + 500
WHERE account.balance > acc.balance
AND b.client.clname = sg.client.cname AND b.client.street = sg.client.street
```

3.4. *MULTIPLE QUERIES*

Multiple queries are queries extending the basic semantics of relational statements to multitables. They are basically formulated as elementary queries, except that the designators refer to multitables instead of tables. The multitables are defined dynamically in the statement through multiple identifiers or semantic variables [1], depending on whether data names are the same or different. The result of a multiple query is a set of tables, unless a built-in function transforms it into a single table. The idea in multiple queries is to express a manipulation broadcast to several databases in a single statement. Such queries have been also called *diffusion queries* or *broadcast queries* [20].

3.4.1. *Multiple Identifiers*

A (multiple) query with multiple identifiers is an equivalent of the set of pertinent (elementary) subqueries resulting from all the substitutions of the multiple identifiers by the corresponding unique ones. The substitutions are in the order of qualification: (multi)database names are chosen first, then the identifiers of tables in the chosen databases are set for the **FROM** clause if any, and finally, the names of columns of the identified tables are determined for the **SELECT** clause. A subquery is *pertinent* when it may be evaluated for the given schema(s). An MSQL subquery is considered pertinent if it designates existing data objects. In particular, the resulting **SELECT** clause must contain only column names of the tables identified for the **FROM** clause (this is an implicit assumption for SQL as well). Other queries may sometimes be rendered pertinent through more or less complex transformations to equivalent pertinent queries [14].

EXAMPLE 3.4. Retrieve all clients of **Banks**:

```
USE Banks
SELECT *
FROM client
```


The designator **client** is a multiple identifier of three tables. The query extends the semantic of the SQL **SELECT** statement to a multitable formed from tables bearing the same name, but not necessarily *union-compatible* as in the example. It replaces three (elementary) SQL queries whose execution order is immaterial. The result is the set of three working tables, each identified by the scope of the subquery [identifier(s) of the database(s) that the tuples come from].

Note that a client may be duplicated, up to three times, but that each tuple then bears obviously different semantic information even if the values are the same.

EXAMPLE 3.5. Update street name "Etoile" to "Charles de Gaulle" in **Banks** and **etoile**.

```
USE Banks etoile
UPDATE *
SET street = 'Charles de Gaulle'
WHERE street = 'Etoile'
```

This statement updates all the tables that have column **street**. It replaces seven SQL **UPDATE** statements. It could replace any number, provided that the cooperating banks agree on the common column name.

3.4.2. *Semantic Variables*

A *semantic variable* is a variable whose domain covers data names in the scope of the query, i.e. names of columns, of tables, or of (multi)databases. The domain may be explicit, which means that names are enumerated in the query. It may also be implicit, which means that it is computed from the variable name through some rules.

The aim of this concept is to enable the user to broadcast his intention over differently named data. 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 be unique or multiple identifiers of some data names. The query is equivalent to the set of pertinent subqueries resulting from possible substitutions of semantic variables and multiple identifiers by unique identifiers. As above, the variables are substituted in descending order of qualification.

The explicit semantic variables are introduced by one or more following clauses:

LET⟨semantic variable⟩**BE**⟨designator⟩[⟨designator⟩]...

⟨semantic variable⟩ :: = ⟨single variable⟩|⟨compound variable⟩
⟨single variable⟩ :: = ⟨character string⟩
⟨compound variable⟩ :: = ⟨single variable⟩.⟨single variable⟩[.⟨single variable⟩]...
⟨designator⟩ :: = ⟨data designator⟩[.⟨data designator⟩]...

The data designators may include the SQL generic characters "%" for any sequence of $n \geq 0$ characters and "_" for any single character. The designator of the form (D.*) means "all columns of table designated by D or all tables of database D etc.". The semantic variable and the designators following it have to be of the same arity. The substitution rules are as follows:

LET x BE city town

x designates **city** or **town**.

LET x.y BE br.t# branch.tel

The couple (x,y) designates either (**br**,t#) or (**branch**,**tel**). Thus the substitution, for instance x ← **br** and y ← **tel** is not allowed. A similar rule holds for triplets, etc.

The implicit semantic variables are assumed to be only character strings with generic characters (note that identifiers may also be considered as trivial implicit semantic variables). Thus the rule for the domain determination is very simple. However, in general, it may be arbitrarily complex, including for instance the recognition that X = "Ville" designates all columns in the scope named "City" or "Town", since the user is French. Implicit variables are used directly in the clauses as are all the other identifiers.

EXAMPLE 3.6. Assuming the scope **Banks**, retrieve all branches at Av. Champs Elysées:

```
LET x BE br branch
SELECT *
FROM x
WHERE street = 'Av. Champs Elysées'
```


First, this query will be decomposed into two queries by substituting **br** and **branch** to x respectively. The query with **branch** will be elementary, but the query with **br** will still be multiple, since **br** is a multiple identifier. This query will lead further to two elementary queries. The result will consist of three elementary queries, to be executed in arbitrary order:

```
USE bnp
SELECT *
FROM br
WHERE street = 'Av. Champs Elysées'

USE sg
SELECT *
FROM branch
WHERE street = 'Av. Champs Elysées'

USE cic
SELECT *
FROM br
WHERE street = 'Av. Champs Elysées'
```

Using the implicit variable, the same query could be formulated as follows:

```
SELECT *
FROM br%
WHERE street = 'Av. Champs Elysées'
```

Both queries extend the semantic of the SQL **SELECT** statement to multitables of tables named differently, which may also have different columns.

EXAMPLE 3.7. Retrieve from **Banks** the names, phone numbers and addresses of branches in department Hauts de Seine (Zip code starting with 92):

```
USE Banks
LET x.y.z.v BE br.tel.street#.city branch.t#.s#.town
SELECT %name, street, z, v, zip%
FROM x
WHERE zip% LIKE '92%'
```

Note the difference between the applications of "%," one to the column name evaluation and the other, classical to SQL, to the column value evaluation. The query generates three SQL queries.

EXAMPLE 3.8. Note in cltype: "Special care" for any **bnp** client who is also a **sg** or **cic** client.

```
USE banks
LET b.t BE sg.ct# cic.cltel
UPDATE bnp.client
SET bnp.client.cltype = 'Special care'
WHERE bnp.client.clname = b.client.%name AND bnp.client.tel = b.client.t
```

The **WHERE** clause addresses a table and a multitable.

3.5. *OPTIONS*

SQL implicitly assumes that all the columns in the **SELECT** statement are in tables in **FROM** clause and, in general, in the schema of the addressed database. It is useful to relax this assumption in the multidatabase environment. The concept of *options* is intended for this purpose. The corresponding syntax is as follows.

Let **d** be a column designator within the **SELECT** statement. Let **q** be a subquery resulting from some substitutions and **a** the unique identifier corresponding to **d** in **q**.

(1) If **d** is preceded by a space, as is usual in SQL, then **q** is only pertinent if there is column **a** in its scope. Thus, by default **a** is mandatory.

(2) **d** written " ~ **d**" means that **q** may be pertinent without a column named **a** in the scope. **q** is then considered as equivalent to a query formulated like **q** without **a** in the **SELECT** list. The column **a** is thus optional.

(3) A list $\mathbf{d}_1|\mathbf{d}_2|\ldots|\mathbf{d}_n$ means that the pertinent form of **q** should contain one and only one $\mathbf{a}_i$. The choice follows the list order. A list preceded with " ~ " means that the whole list is optional.

EXAMPLE 3.9. Create a new account for Durand D. in the branch database **etoile** and in **bnp**:

```
USE bnp etoile
INSERT INTO a% (a%, c%, ba%, ~ br%#, ~ open_date)
VALUES (123, 456, 789, 010, 11/04/87)
```

The value of **br#** will be inserted only into **bnp**, that of **open_date** only into **etoile**. If these columns were not optional, none of the insertions would be made, since both subqueries would not be pertinent. The implicit semantic variables were useful only to shorten the query expression.

EXAMPLE 3.10. Select from **Banks** the account number, client number, balance and opening date, if any, of all accounts:

```
USE Banks
LET x BE acc%
SELECT ac%#, c%#, balance, ~ open_date
FROM x
```

Since the column **open_date** is optional, all three databases will be addressed. If **open_date** were mandatory, the tuples would be retrieved only from the **bnp** and **sg** databases.

EXAMPLE 3.11. Assume that **sg** database does not have the column **t#** in the table **branch**. Retrieve from **Banks** branch names and either only the phone numbers if available or the corresponding streets:

```
USE Banks
LET x BE br%
SELECT %name, t%|street
FROM x
```

The query will provide the phone number from **bnp** and **cic**, and the street from **sg**.

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

3.6. *OUTER JOINS*

Inner joins eliminate the unmatched rows. MSQL makes it possible to keep such rows through **outer joins**, like Oracle-SQL and Transac-SQL [26]. This possibility is fundamental in the multidatabase environment [10]. The outer join is symbolized with either "?" or "*," the latter standing for *natural outer join*. The symbol precedes or follows a usual comparison operator; let it be **ß**. Thus:

(1) a left outer join is denoted **?ß** or ***ß**,
(2) a right outer join is denoted **ß?** or **ß***,
(3) a full outer join is denoted **?ß?** or ***ß***.

When outer join clauses are used, the order of clauses in the **WHERE** statement is in general important. The usage of the outer join is subtle; see for instance [7] or [26] for discussion.

EXAMPLE 3.12. Select the name, phone number, and balance of all **bnp** clients if their balance exceeds 1,000,000 FFr, and their **sg** balance if any:

```
USE (bnp b) sg
SELECT clname, cltel, account.balance acc.balance
FROM b.client, account sg.client
WHERE  b.client.cl# = account.cl#  AND  b.client.clname  *=
  sg.client.cname
  AND b.client.cltel *= sg.client.ct#
  AND account.balance > 1000000
```

3.7. *COLUMN LABELS*

As in most SQL dialects, a column designator in the MSQL **SELECT** statement may be followed by a label. The label then heads the corresponding column values in the query result. In particular, this possibility is useful for homogenizing column names as a result of a multiple query. If there is no label, the current column name is displayed.

EXAMPLE 3.13. Select the names, phone numbers, and street numbers of all branches at Champs Elysées. Present them with unified headers: **branch_name**, **tel#**, and **street#**:

```
USE Banks
LET t BE tel t#
SELECT %name branch∘name, t tel#, s%# street#
FROM br%
WHERE street = 'Champs Elysées'
```

3.8. *UNION*

The **UNION** operator eliminates duplicates from union-compatible tables and makes them a single table. The corresponding syntax is as follows:

```
⟨SELECT statement⟩[UNION⟨SELECT statement⟩]... |
⟨SELECT statement⟩UNION *
```

The **SELECT** statements in the first statement must define tables. The latter statement unions all tables produced by the subqueries of a multiple query, provided they are all union-compatible.

EXAMPLE 3.14. Select from **bnp**, **sg**, and **cic** the names, phone numbers, and addresses of all clients:

```
USE Banks
LET x.y.z BE clname.cltel.street# cname.ct#.s#
SELECT x client_name, y client_tel#, street client_street, z street_#
FROM client
UNION *
```

The result will be a single table with the labels as column headers.

3.9. *BUILT-IN FUNCTIONS*

3.9.1. *Extended SQL Functions*

SQL has some built-in functions, namely **COUNT**, **AVG**, **MAX**, **MIN**, and **SUM**. MSQL extends the use of these functions to elementary multidatabase queries and to multitables. In the former case, the function is used under its SQL name. In the latter case two situations may occur:

(1) The function aggregates each table of the multitable. It then keeps its original SQL name.

(2) The function aggregates the whole multitable. It then has the same syntax, except that its name is prefixed with the letter **M**. For instance, **AVG** becomes **MAVG**.

In particular, one can use either **DISTINCT** or **MDISTINCT**.

EXAMPLE 3.15. Retrieve the average balance per bank (i) and then the average balance of all banks (ii):

```
USE Banks                                   (i)
SELECT AVG(ba%)
FROM a%

SELECT MAVG(ba%)                            (ii)
FROM a%
```

EXAMPLE 3.16. To find the concentration of millionaires near the Place de l'Etoile area, retrieve the number of millionaires per **bnp** and **etoile** (i) and then

the total number of millionaires (ii):

```
USE bnp etoile                                              (i)
LET x BE bnp etoile
SELECT COUNT(DISTINCT client.cl#)
FROM x.client x.account
WHERE x. client.cl# = x.account.cl#
AND x.account.ba% > 1000000

SELECT MCOUNT (MDISTINCT client.cl#)                        (ii)
FROM x.client x.account
WHERE x. client.cl# = x.account.cl#
AND x.account.ba% > 1000000
```

The keywords **DISTINCT** and **MDISTINCT** are needed, as a millionaire may have more than one large account.

EXAMPLE 3.17. Retrieve the list of clients respectively from **bnp** and **sg** banks grouped (i) by the branch they belong to and (ii) by the client type only:

```
USE bnp sg                                                  (i)
SELECT c%name
FROM client x, br% y
GROUP BY br%name
HAVING x.c%# = acc%.c%#
AND acc%.br%# = y.br%#

SELECT c%name                                               (ii)
FROM client
MGROUP BY c%type
```

3.9.2. *Specific Functions*

The analysis also showed the need for specific new built-in functions [17]. The following functions should be found particularly useful.

MERGE This function, called **NORM** in [17], merges into one row all the rows which correspond to the same real object in different relations of a multiple query. The merge uses the natural outer join over the designated

columns. Its syntax is

MERGE ON⟨column designator⟩[,⟨column designator⟩]...

EXAMPLE 3.18. Select from all banks the accounts of all millionaires. If the client has an account in more than one bank, then merge the corresponding data:

```
USE Banks
LET x.y BE clname.cltel cname.ct#
LET z BE Banks.*
SELECT *
FROM z.a%
WHERE z.a%.c%# = z.client.c%#
AND z.a%.balance > 1000000
MERGE ON x y
```

NAME 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 (table for a column, 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 a column name within **SELECT** and **WHERE** statements. The result is then considered as if the value of the column were the corresponding character string.

EXAMPLE 3.19. Retrieve the names of branches in Nice and of the banks they belong to:

```
USE Banks
LET x.y BE br.city branch.town
SELECT %name, NAME(.x)
FROM x
WHERE y = 'Nice'
```

It can be seen that database names may bear information in the multidatabase environment.

UPTO This function limits the duplication of information that may come from several databases. For instance, a query to ten bank databases may at most ask for two references of a client. In particular the user may give priority

to databases he trusts more than others. The function syntax is as follows:

UPTO (n(⟨A⟩)[⟨B⟩])

⟨**A**⟩ :: = ⟨column list⟩
⟨**B**⟩ :: = ⟨database list⟩

The function provides at most **n** − 1 tuples sharing the values of columns 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 nonnull response from **n** databases. The default value of **n** is 1. Finally, empty **B** means that the user has no preference.

EXAMPLE 3.20. Retrieve five branches at Champs Elysées, preferably **bnp** branches if not **cic**.

```
USE Banks
UPTO (5 (street) [bnp cic])
SELECT *
WHERE br%.street = 'Champs Elysées'
```

The result will come from the **cic** database only if there are less than five **bnp** branches on the Champs Elysées. If both banks have less than five branches at this location, then **sg** will be searched.

3.10 INCOMPLETE QUERIES

When formulating MSQL 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* [15]. They are deduced by the system from database schemas. The result is called the *complete query*. The completion algorithm is described in detail in [16]. It is shown that this process leads to the intuitively expected result in more cases than the present algorithms for the universal relation interface [27]. A major consequence is also that updates may be performed.

The goals of this function are (1) to further simplify query formulation, and (2) to make the expression of a multiple query independent of differences in decomposition into tables. Otherwise, there is sometimes no way of formulating a single statement, if one has to indicate all equijoins corresponding to different decompositions.

EXAMPLE 3.21. Retrieve from **bnp** the addresses of all branches where Dupond has an account.

The incomplete query could be

```
Use bnp                                          (3.21.1)
SELECT street
WHERE clname = 'Dupond'
```

The complete query would be

```
USE bnp                                          (3.21.2)
SELECT street
FROM br x, account y, client z
WHERE z.clname = 'Dupond'
AND z.cl# = y.cl#
AND y.br# = x.br#
```

EXAMPLE 3.22. Consider now that instead of three tables, **cic** contains only one (universal) table with all columns in **cic**. Assume further that the user wishes to broadcast the query about Durand to both **bnp** and **cic**. The formulation (3.21.1) 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 (3.21.2) and the query

```
USE cic
SELECT street
WHERE clname = 'Durand'
```

See also Example 5.4.

3.11. DYNAMIC COLUMNS

Similar data in autonomous databases may differ with respect to value types or meaning and may have a highly fluctuating relationship. The balance may be expressed in US$ in one database and in FFr in another database, with the exchange rate varying constantly. Nevertheless, one may need to specify a join on such data, while the comparison between values in US$ and in FFr is meaningless, or one may need to convert all the selected values to the same currency, etc. The multidatabase manipulation languages should therefore provide functions for the corresponding needs.

The concept of *dynamic attribute* is intended for such situations [16]. To match SQL terms, we speak in MSQL about *dynamic columns*. The user defines a dynamic column, say **D**, in a query. He then may use **D** as an actual column, except sometimes for updates. Unlike an actual or a virtual column, **D** is however unknown to the database schema or to any view schema and disappears with the query or at the end of the session. To define **D**, one specifies the *retrieval mapping*, say $\mathbf{M}:\mathbf{C} \rightarrow \mathbf{D}$, for the conversion of the actual values of *source columns* **C** to the dynamic ones, and/or an inverse *update mapping* $\mathbf{M}':\mathbf{D} \rightarrow \mathbf{C}$, if **D** encounters updates. The system may be able to automatically determine one of the mappings given the other, as does MRDSM [17]. See [16] for details of possibilities the concept of dynamic attribute opens. This concept corresponds also to some of the proposals in [6].

A dynamic column is defined by a **D-COLUMN** clause. Its syntax is as follows:

D-COLUMN [HOLD] ⟨definition⟩

⟨definition⟩ :: = ⟨auto-conversion⟩|⟨user-specification⟩
⟨auto-conversion⟩ :: = (⟨source column⟩)⟨D-name⟩**FROM**⟨source unit⟩**TO**⟨target unit⟩
⟨user-specif⟩ :: = (⟨source column⟩[, ⟨source column⟩]...)⟨D-name⟩[= ⟨retrieval mapping⟩] [**AND** [⟨update column⟩] [= ⟨update mapping⟩]]
⟨mapping⟩ :: = ⟨arithmetical formula⟩|⟨SQL string function⟩|⟨SQL date function⟩|⟨table⟩
⟨table⟩ :: = (⟨source value⟩[, ⟨source value⟩]..., ⟨target value⟩[, ⟨target value⟩]...)
[(⟨source value⟩[, ⟨source value⟩]..., ⟨target value⟩[, ⟨target value⟩...)]...

The D-name is the name given to the dynamic column; let it be **D**. The keyword **HOLD** means that **D**'s definition holds for further queries, otherwise **D** may be referred to within the query only. The auxiliary command **LIST HOLD** lists the set of hold names, and **DROP HOLD** ⟨D-name⟩ drops the specified column from the corresponding catalog. See [19] for deeper discussion of the **HOLD** option.

The ⟨auto-conversion⟩ automatically defines the unit conversion mappings, when the system supports it [19]. Otherwise, ⟨user-specif⟩ is the user definition of **D**. The update (retrieval) mapping may be omitted if one performs a retrieval (an update) only or when the system may deduce it from the other mapping [19]. The update column has to be indicated when the retrieval mapping maps several source columns to **D** and does not by itself allow the deduction of the

source values the **D** update should modify. Consider for instance the mapping

month_sal = = nb_days * day_sal.

EXAMPLE 3.23. Consider that **citybank** has a database like **bnp**, except that the balance is in US$. For **bnp** clients whose **bnp** amount is greater than **citybank** amount, retrieve client names and balances in US$, according to the exchange rate of the hour, that is, 6.5 FFr/US$. Order the result by the converted value. Then, add to each **bnp** client the fidelity bonus of 10 US$:

```
USE (citybank c) (bnp b)                                              (3.23.1)
D-COLUMN HOLD (b.balance)
balance = b.balance / 6.5 AND balance = balance * 6.5
SELECT b.clname balance
FROM b.account, b.client
WHERE balance > c.balance
AND b.client.cl# = b.account.cl# AND b.client.clname = c.client.clname
AND c.client.cl# = c.account.cl#
ORDER BY balance

UPDATE account                                                        (3.23.2)
SET balance = balance + 10
WHERE balance > c.balance
AND b.client.cl# = b.account.cl# AND b.client.clname = c.client.clname
AND c.client.cl# = c.account.cl#
```

The query (3.23.2) uses the scope and the dynamic column of (3.23.1). The update mapping may be omitted if the system is able to deduce it. The dynamic column name has priority in the case of name conflict, as in these queries. The query (3.23.1) may be expressed in SQL using a value expression a view or a derived table, but each time in a much more procedural form. The query (3.23.2) cannot be expressed.

EXAMPLE 3.24. Assume that like MRDSM, the system is able to find inverse mappings if the retrieval mapping is a polynomial formula. Consider that the client address is one column with client town, street, etc. Retrieve the name, the town and balance in US$ of clients that do not live in Paris or Marseilles and the value of a variable $x = 2*balance^2 + 2*balance$, if $x = 24$. Order the result by town and ascending balance. Then set $x \leftarrow 40$.

Using the dynamic columns, the query may be expressed as follows:

```
USE bnp                                                    (3.23.3)
D-COLUMN HOLD (addr)
town = LTRIM (RTRIM (SUBSTR (addr, (INSTR( addr, ' '))),' '),' ')
D-COLUMN (balance) x = 2*balance**2 + 2*balance
SELECT town, clname, balance
FROM account, client
WHERE ( town < >'PARIS') AND ( town < >'MARSEILLE')
  AND ( x = 24 ) AND account.cl# = client.cl#
ORDER BY town, balance ASC

UPDATE account                                             (3.23.4)
SET x = 40
WHERE ( town < >'PARIS') AND ( town < >'MARSEILLE')
AND ( x = 24 ) AND account.cl# = client.cl#
```

It is instructive to find the SQL expression of (3.23.3). The query (3.23.4) illustrates the interesting problem that the retrieval mapping above has potentially two inverse mappings corresponding to the solutions of the equation

$$2*balance^2 + 2*balance - 24 = 0$$

MRDSM chooses the inverse formula that corresponds to the actual value *before the update*. Thus if this value is 3, the update will lead to **balance** = 4, else **balance** = −5.

4. VIEWS

4.1. *VIEW CREATION*

A view in MSQL is a virtual table defined by the **CREATE VIEW** statement. As in SQL, this statement simply names and stores a selection expression. The query involving the view is processed through the query modification principle, extended to the multidatabase environment. To create a view may be useful when a query is frequent and complex or for security reasons. A multidatabase view is a view whose selection expression refers to several databases.

The statement form is as follows:

CREATE VIEW ⟨view designator⟩[(⟨view column list⟩)] **AS** ⟨select statement⟩
[WITH CHECK OPTION]

⟨view designator⟩ :: = [⟨database designator⟩.]⟨view name⟩
⟨view column list⟩ :: = ⟨column name⟩[, ⟨column name⟩ . . .]

Any view belongs to some databases, which may be virtual. The statement creates view(s) named upon the ⟨view name⟩ either in each designated database or in each database in the current scope. To designate several databases one should use a semantic variable ranging over the desired database names in the scope. If the view column list is used, then *i*th view column name renames the *i*th column in the selection expression (if "*" is used, then the order is that of the corresponding **CREATE TABLE** statement). Otherwise, the view columns are named upon those in the selection expression. The selection expression may include all the new functions, provided the result is a table (and not a multitable). It may also refer to other views. If the **WITH CHECK OPTION** is specified, then the created views may be updated.

In SQL, a view is implicitly a view of the database used and in this database. These notions should be carefully distinguished while using MSQL, where the view of one or more database may be created in different and explicitly named (one or more) databases. To create a view in a database **B** basically means that the view is stored in **B** catalogs. The view is created in **B** only if **B** accepts it. An actual database may refuse a user view if it is a public database or if system catalogs refuse references to tables in other databases. Such views, and indeed any view, may be included in a virtual database, in particular, that of the view creator. The concept of the virtual database will be discussed in the next section.

EXAMPLE 4.1. Create in a virtual database **my_bank** the view **same_street_branches** of **bnp** and **sg** branches at the same street:

```
USE my_bank bnp sg
CREATE VIEW my_bank.same_street_branches
(bnp_name, bnp_s#, sg_name, sg_s#, street, city)
AS SELECT brname, street#, braname, s#, street, city
FROM br, branch WHERE br.street = branch.street AND br.city =
branch.town
```

This view is a multidatabase view.

EXAMPLE 4.2. Create in **bnp** and **sg** views **account$** of selected data about client accounts in **Banks**, with homogenized naming and balance updatable and expressed in US$ instead of FFr:

```
USE Banks
LET x BE bnp sg
CREATE VIEW x.account$ (cl#, name, street, acc#, balance$)
AS LET y BE Banks.*
   D_COLUMN (balance) balance$ = balance/5.95
   SELECT c%#, c%name, street, ac%#, balance$
   FROM y.a%, y.c%
   WHERE y. a%.ac%# = y.c%.ac%#
```

4.2. *VIEW REMOVAL*

MSQL makes it possible to drop one or several views using the **DROP VIEW** statement, similar to the **DROP TABLE** statement:

DROP VIEW ⟨view designator⟩[⟨view designator⟩...]

4.3. *VIRTUAL DATABASES*

A *virtual database* is a named set of views. One can use it as an actual database, except for eventual restrictions on updates. It may be useful for the following purposes:

(1) As an external schema (in ANSI/SPARC) on an actual database, to define the conceptual schema for the multidatabase usage, also called the *import schema* in [12]. It may hide tables one wishes to keep private, rename tables and columns to the common naming schemes, etc.

(2) As the user perception of one or more databases, through mono- or multidatabase views. The database is then the import schema in the sense of [12] and a global schema in the sense of INGRES-STAR [13]. As virtual data constitute a logically single database, SQL may suffice for data manipulation, though new capabilities of MSQL may reveal helpful anyhow.

A virtual database is created, altered, or dropped using the corresponding **CREATE DATABASE, ALTER DATABASE,** and **DROP DATABASE** statements.

5. INTERDATABASE QUERIES

5.1. *INTRODUCTION*

Traditional queries basically transfer data between a database and the user workspace. An *interdatabase query* transfers data between *databases* [23,24]. The need for interdatabase queries often exists at present, especially when mainframe databases cooperate with those on workstations. The corresponding MSQL statements copy or move data from one or more *source tables* to one or more *target tables*. Both kinds of tables may be in different databases. The transfer may add only tuples, but may also import some source column schemas and values, all in an atomic statement. This capability makes some interdatabase queries much less procedural than if the traditional separation of data and schema operations into distinct statements were enforced.

MSQL provides for interdatabase queries new possibilities for the **INSERT** and **UPDATE** statements, as well as specific statements termed **STORE, REPLACE,** and **COPY.** For all statements, source data are selected using a selection expression including the possibilities discussed up to now and other specific ones discussed below.

Different statements express different interactions between incoming and existing data, especially when the keys match. One may indeed wish to preserve the existing tuple, whereas another application may need the incoming tuple to replace the existing one, or one may even wish to replace the whole content of the target table. Furthermore, the statements provide new facilities for data conversion, as incoming data will usually need to fit a preexisting and different schema and may need to go to different schemas simultaneously. The conversion may concern (i) column value representation, units of measure or meaning (price without taxes to prices with taxes), (ii) column order or names, and (iii) table names. MSQL assumes the value representation conversion to be automatic. Value expressions, dynamic attributes, and built-in functions are available for units and meanings conversions. Functions for conversions (ii) and (iii) are described below.

5.2. **INSERT** *STATEMENT*

This statement adds rows to target tables and may add columns. The converted row is inserted if its key does not match the existing one (if no key is indicated, all existing target columns are considered). Otherwise, only the values of the new columns are transferred. For other existing rows, these values are set to null. If the selection expression is multiple, the union of all subqueries'

results must provide only one row by key value. Otherwise, one cannot not know which row should be inserted.

The general form of the **INSERT** statement is as follows:

```
[⟨USE statement⟩] [⟨LET statement⟩]                                    (5.1)
INSERT [INTO] [⟨target table designator⟩]|[(⟨target column list⟩)]
[WITH CONSTRAINTS] ⟨source selection statement⟩

⟨source selection statement⟩ :: =
[⟨USE statement⟩]
[⟨LET statement⟩]
SELECT|MOVE ⟨source column designators⟩
[⟨FROM clause⟩]
[⟨WHERE clause⟩]
```

The target may be a multitable. **MOVE** deletes the source tuples the selected values come from. The possibilities for column name and order matching are basically of two kinds:

By order (default option). The *i*th selected column, either in the source list or in the **CREATE** statement of the source table if the source list is "*", is mapped on the *i*th target column in either target column list or in the **CREATE** statement of the target table. Each list may include characters "$", meaning that the corresponding source or target column is skipped.

By name. Each selected column is mapped on the target column with the same name or with the name of the source column label, if used. A label is mandatory with a source value expression. This possibility requires either "**" or "* = *" in the target specifications, as shown in the examples below, or at least one label in the source list. In particular, the target table designators may be omitted, the tables then being selected through the column name matching, regardless of table names. When target key columns do not match, the target table is not involved.

The mapping by order (without the "$" possibility) is already used by the SQL **INSERT** statement. For multiple target tables, it provides independence with respect to target column names. The mapping by name is new and provides independence with respect to the target column order. It further provides independence with respect to the decomposition of incoming columns into target tables. This type of mapping is closer to the spirit of the relational model, since the columns basically have no order. In the multidatabase environment, it is also frequently more useful. In particular, it limits mapping specification errors, since column lists are shorter and not redundant.

The options may be used in both source and target lists. In addition to those already defined, the option "+a" means that if the target column **a** does not exist, then it is created. If the **WITH CONSTRAINTS** option is used, then **a** inherits any constraint attached to its source column, unless it contradicts the target data. If there is no "+a" option, then for any target table the only source values effectively mapped are those whose target columns exist. The choice may depend on the table if the tables designated by the target table designator differ with respect to the columns. The "+" option may be nested with others, but only as the last element (e.g. **a|b|c|+a**).

The column lists may contain elements of the form **a:b**. They mean that all the columns between **a** and **b** are implicitly involved in the order of the **CREATE TABLE** statement, unless a column is explicitly designated elsewhere in the statement. By default, **a** is the first column and **b** the last one.

EXAMPLE 5.1. Add into **bnp** new clients from **etoile**:

```
USE bnp etoile
INSERT bnp.client ( :cltel, street: )
SELECT * FROM etoile.client
```

The mapping is by order. In all new rows, **cltype** is set to null value.

EXAMPLE 5.2. Move to **bnp.account** data in **open_date** column and all new accounts in **etoile, opera** and **nation** branches:

```
USE bnp etoile opera nation
INSERT INTO bnp.account ( :balance, +open_date )
LET x BE etoile opera nation
MOVE * FROM x.acc%
```

The column **br#** will be set to null in new target rows. The query is equivalent to one **ALTER TABLE** statement and three elementary inter-database **INSERT** statements.

5.3. STORE *STATEMENT*

The only difference from the **INSERT** statement is that **STORE** refreshes all existing columns corresponding to incoming ones when the keys match. The target columns that do not correspond to incoming ones remain unchanged. The general form of the **STORE** statement is like (5.1) except that the command **STORE** replaces **INSERT**.

EXAMPLE 5.3. Refresh in **bnp** new balances of all branches and add all new accounts. Then, assuming some processing of the **bnp** accounts, which in real life is usually done overnight, send back all processed accounts:

```
USE bnp etoile opera nation
STORE bnp.account
LET x BE etoile opera nation
SELECT y.:balance, br.br#
FROM x.acc% y, br
WHERE br.brname = NAME(x)

USE bnp etoile opera nation
LET y BE etoile nation opera
STORE y.acc%
USE bnp
SELECT acc# %c#, balance
FROM account br
WHERE account.br# = br.br# AND br.brname = NAME(y)
```

The target **open_date** values will be preserved, since no incoming column is mapped onto them. Note the use of "%" in the label; one could also apply the option **acc# | nac#**.

5.4. REPLACE *STATEMENT*

This statement erases all existing target rows and then inserts the incoming ones. The general form of the statement is like (5.1) except for the command itself.

EXAMPLE 5.4. Replace all clients and accounts of **etoile** and **nation** with those of **bnp**:

```
USE etoile nation
LET db BE nation etoile
REPLACE (**)
  USE bnp
  SELECT y.*, x.:, cltel tel
  FROM client x, account y, br
  WHERE x.cl# = y.cl# AND y.br# = br.br# AND br.brname =
    NAME(db)
```


The label is necessary for **cltel** mapping, since mapping by name is used. The source columns **cltype** and **br#** are not mapped, despite being selected. The query is a multiple interdatabase query leading to four elementary queries.

5.5. *UPDATE STATEMENT*

The statement does not create new rows, but only modifies existing columns using the values of some source columns. The mapping results from the selection expression that has the **FROM** clause, unknown to the SQL **UPDATE** statement, except in [26]. The statement may copy or move the incoming values.

EXAMPLE 5.5. Update **bnp.client.cltel** with the values from **etoile, nation,** and **opera** DBs:

```
USE bnp etoile opera nation
UPDATE bnp.client
  LET x BE etoile nation opera
  SET ctel = %tel
  FROM x.c%
  WHERE bnp.client.cl# = x. c%.%cl#
```

5.6. COPY *STATEMENT*

The **COPY** statement duplicates selected source rows and schemas. It also duplicates the corresponding source constraints if the **WITH CONSTRAINT** option is used and the constraints are preserved by the selection expression. The selection may involve value conversion and may change column names or order. The general form of this statement is like (5.1) except for the statement command.

EXAMPLE 5.6. Copy to a new branch database **versailles** all corresponding accounts in **bnp**, creating the table **acc(nacc#, ncl#, balance)**:

```
USE (versailles v)
COPY v.acc
  USE bnp
  SELECT a.acc# nacc#, a.cl# ncl#, a.balance
  FROM account a, br
  WHERE a.br# = br.br# AND br.city = 'versailles'
```

EXAMPLE 5.7. Save **bnp** tables **account** and **client** into **tables a** and **c** in **save _bnp**:

```
USE bnp save_bnp
LET y.x BE a.account c.client
COPY y
  SELECT *
  FROM x
```

6. AUXILIARY DATA OBJECTS

6.1. *MANIPULATION DEPENDENCIES*

Manipulation dependencies specify manipulations of some tables triggered by a given manipulation of a given table. They are manipulated using the well-known concept of triggers. Triggers are defined using the following statement, similar to that of [26]:

CREATE TRIGGER⟨trigger name⟩**ON**⟨table designator⟩⟨trigger definition⟩

⟨trigger definition⟩ :: = **FROM** ⟨trigger identifier⟩|**FOR**{**INSERT**|**UPDATE**|**DELETE**},...,**AS** ⟨program⟩

One may define any number of triggers on a table, provided their names differ. The table designator may designate a multitable, provided the tables are in different databases. The statement then creates the same trigger on each table. The program may be an arbitrary program or MSQL statements with, possibly, flow control statements **if, else,** etc. It may include the following statement that tests the update of the given column(s):

IF UPDATE (⟨column designator⟩)[{**AND**|**OR**}⟨column designator⟩],...⟨program⟩

Finally the program may refer to two dynamic tables **inserted** and **deleted** [26]. The tuples of **inserted** are copies of those inserted or of these already updated by the statement that fired the trigger. Tuples in **deleted** are copies of those deleted from the table or of those updated, but with the before-update values.

The table designator may in particular refer to a table that may trigger a manipulation of another trigger table, etc. This possibility should be used carefully, as it may lead to loops or infinite executions. An implementation may also put limitations on the statement capabilities [26]. The trigger is removed through the statement

DROP TRIGGER ⟨*trigger identifier*⟩

EXAMPLE 6.1. The banks have decided to create in each database a table where data are diffused to prevent fraud:

crooks (bank, clname, acc#)

It was further decided that any insertion should go to any table, in order to allow any selection to use only the local table (thus **crooks** tables are fully duplicated). The following trigger makes it possible to designate only the local table, for both insertions and selections:

INSERT crooks FROM inserted

The table **inserted** is then automatically always that of the database whose **crooks** table is designated in the **INSERT** statement that fired **warning**.

6.2. *EQUIVALENCE DEPENDENCIES*

These dependencies declare the columns whose equal values identify the same real object. They are particularly useful for the calculus of the implicit interdatabase joins [15]. A column may in particular be a virtual one, defined to provide the value equality when the actual columns modeling the same object differ in units, precision, etc. This column may also combine several actual columns. The corresponding statement is as follows:

CREATE LINK ⟨link name⟩ (⟨column designator⟩ = ⟨column designator⟩
[**AND**⟨column designator⟩ = ⟨column designators⟩]...
[,⟨column designator⟩ = ⟨column designators⟩...]...)

The link is created for the databases in the scope. The **AND** clause indicates columns to be considered together, for instance client name and phone number, all columns on the same side of " = " being then in the same table. The commas separate parts of the dependency, for instance the equivalence between the

databases **etoile** or **bnp** and **nation**, and then that between **nation** and **opera** (see the example below). Parts and separate links may lead to transitivities if they share columns. The link may be altered or dropped using the corresponding statements with obvious syntax.

EXAMPLE 6.2. Declare the equivalences of keys of clients and of accounts in BNP databases:

```
USE bnp opera etoile nation
CREATE LINK same_client (cl# = %cl#)
CREATE LINK same_account (acc# = nation.acc#, nation.acc# =
  opera.nac#)
```

The link **same_account** allows, for instance, the query asking for selection of **bnp.br#** and of **etoile.open_date** to omit in the **WHERE** clause the interbase join on **acc#**.

6.3. *STORED QUERIES AND TRANSACTIONS*

Any MSQL query or transaction may be named, stored in source-compiled form, and then recalled for execution. One should use for this purpose the **PROCEDURE** statement, whose syntax is

PROCEDURE ⟨procedure name⟩⟨parameters⟩⟨MSQL statements⟩

The semantics of the statement and the usage of procedures are as in SQL in [25] except that (1) MSQL statements may be used and (2) the procedure name may be a multiple identifier. In the latter case, the statement duplicates the procedure in all the databases in the scope, providing the appropriate privileges.

7. CONCLUSION

As databases are becoming easily accessible on computers and networks, users face multiple and autonomous databases. New functions for data manipulation languages are then needed, as the present languages were designed for manipulations of a single integrated database. MSQL offers such functions for relational databases. Numerous examples have shown that the proposed functions should prove useful for a large variety of user needs. This is due to their flexibility and openness with respect to the accommodation of autonomous

names, values, and structures. Unlike the classical database languages designed for a single truth, they are intended to face not only multiple different truths, disagreements, and contradictions, but also cooperations, which together are the most exciting aspects of real life. Most of these functions are not yet available in other languages and systems.

MSQL extends the possibilities of SQL to a multidatabase environment and the proposed functions are designed for this environment. Several functions have nevertheless proved useful in a single database context. Thus, the concept of the multiple identifier may help in preserving 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.

The proposed functions lead to many open problems at the implementation level. Also, further functions may be added. Knowledge processing techniques should be investigated, as they seem particularly interesting in the multidatabase environment [11,21]. 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. Finally, one may build upon MSQL a graphical interface like that in [5].

APPENDIX. EXAMPLE DATABASES

The databases used throughout the examples are supposed to be databases of the French banks BNP, Société Générale, and CIC. The databases **bnp, sg,** and **cic** are the main databases of the corresponding banks, on mainframes in real life. The databases **etoile, opera,** and **nation** are databases of BNP branches, named for their location inside Paris and used for local processing. They contain only data related to their customers and exchange them back and forth with the main database, usually during the night. Some of their clients or accounts may be unknown to **bnp**, at least temporarily.

DB **bnp**:

br (br#, brname, street, street#, city, zipcode, tel)
account (acc#, cl#, balance, br#)
client(cl#,clname,cltel,cltype,street,street#,city,zipcode)
spe-acc(acc#,br#,cl#,balance,cur)

DG **sg**:

branch (bra#,braname,street,s#,town,zip,t#,class)
acc (acc#,bra#,c#,balance)
client (c#,cname,ct#,ctype,street,s#,town,zip)

DB **cic**:
br (br#, brname, street, street#, city, zipcode, tel)
account (ac#, br#, cl#, balance, open ∘ date)
client(cl#, clname, cltel, cltype, street, street#, city, zipcode)

DB **etoile**:
account(acc#, cl#, balance, open_date)
client(cl#, clname, tel, street, street#, city, zipcode)

DG **nation**:
acc(acc#, cl#, balance, open_date)
client(cl#, clname, tel, street, street#, city, zipcode)

DB **opera**:
account(nac#, ncl#, balance, open_date)
customer(ncl#, cname, ctel, street, street#, city, zipcode, ctype)

The tables within the main databases model respectively branches, clients, and accounts in FFr. The table **spe-acc** in **bnp** contains accounts in foreign currencies, defined in the **cur** column. The underlined columns are keys. The databases are to some extent semantically heterogenous with respect to names, structure, and value types, despite being all relational databases. This is because they are autonomous, as banks and branches compete for clients. Nevertheless, the banks and branches also cooperate, the cooperation being naturally stronger between branches of the same bank. That is why data in different databases have also some similarities which are stronger inside the same bank. The whole situation leads to the following assumptions for the example:

(1) Banks partly disagree upon the names which model the same concepts.
(2) The same client may be respected in several databases.
(3) Primary key values in databases of different banks are independent, despite sometimes having the same column names.
(4) However, they are the same for the same client, account or branch, inside BNP databases.
(5) Finally, the banks consider that the same name and address or phone number identify the same client in any database.

The authors are grateful to the referees for various suggestions and to R. James for editorial help.

REFERENCES

1. A. Abdellatif, Multiple queries in the multidatabase system MRDSM (in French), Thesis Univ. of Tunis, INRIA, June 1983, p. 80.

2. ANSI-SPARC, Interim Report on Database Management Systems, Doc. 7514TS01, Washington, Feb. 1975.
3. S. Ceri and G. Pelagatti, *Distributed Databases: Principles and Systems*, McGraw-Hill, New York, 1984.
4. E. F. Codd, A database sublanguage founded on the relational calculus, *ACM SIGFIDET*, Nov. 1971, pp. 35–68.
5. B. Czejdo, M. Rusinkiewicz and D. Embley, A Unified Approach to Schema Integration and Query Formulation in Federated Databases, Res. Rep., Univ. of Houston, 1987, p. 25.
6. C. J. Date, *A Guide to DB2*, Addison-Wesley, 1983.
7. C. J. Date, The outer join, in *2nd International Conference on Databases (ICOD-2)* (S. M. Deen and P. Hammersley, Eds.), Wiley, 1983, pp. 76–106.
8. C. J. Date, A critique of the SQL database language, *ACM SIGMOD Record* 14, No. 3 (Nov. 1984).
9. C. J. Date, *An Introduction to Database Systems*, 4th ed., Addison-Wesley, 1986.
10. U. Dayal and M. G. Gouda, Using semiouter joins to process queries in multidatabase systems, in *ACM-PODS*, Waterloo, Canada, Apr. 1984, pp. 153–162.
11. Li Deyi, *A Prolog Database System*, Research Study Press, England, 1984.
12. D. Heimbigner and D. McLeod, Federated architecture for information management, *ACM Trans. on Office Inform. Syst.* 3(3):253–278 (1985).
13. *INGRES-STAR User's Guide*, Relational Technology Inc., 1986.
14. W. Litwin et al., Sirius systems for distributed data management, in *Distributed Databases* (H. Schneider, Ed.), North-Holland, New York, 1982, pp. 311–366.
15. W. Litwin, Implicit joins in the multidatabase system MRDSM, presented at IEEE-COMPSAC, Chicago, Oct. 1985.
16. W. Litwin and P. Vigier, Dynamic attributes in the multidatabase system MRDSM, presented at COMPDEC, Los Angeles, Feb. 1986.
17. W. Litwin and A. Abdellatif, Multidatabase interoperability, *Computer* 19(12):10–18 (Dec. 1986).
18. W. Litwin and A. Abdellatif, An overview of the multidatabase manipulation language MDSL, *Proc. IEEE*, May 1987, p. 12.
19. W. Litwin and Ph. Vigier, New capabilities of the multidatabase system MRDSM, in *Proceedings of the XLV Forum of HLSUA*, New Orleans, Oct. 1987, pp. 467–476.
20. P. Lyngbaek and D. McLeod, An approach to object sharing in distributed databases, in *VLDB 83*, Florence, Oct. 1983, pp. 364–376.
21. C. Marcus, *Prolog Programming*, Addison-Wesley, 1986, p. 325.
22. *MRDS Multics Relational Data Store, Reference Manual*, CII HB, Jan. 1982.
23. B. Nicolas, A. Zeroual, and W. Litwin, Interdatabase queries in multidatabase systems (in French), in *Journées Internationales de l'Informatique et de l'Automatisme — JIIA 86*, ed. JIIA, Paris, 17–20 June 1986, pp. 233–243.
24. B. Nicolas and W. Litwin, Interdatabase queries in multidatabase systems, presented at IEEE Computer Society's Workshop on Design Principles for Experimental Distributed Systems, Purdue Univ., West Lafayette, Ind., 16–17 Oct. 1986.
25. *Information processing systems — Database language SQL*, Draft International Standard ISO/DIS 9075, p. 114.
26. Sybase Database Management, *Transact-SQL User's Guide*, Doc. 3231-2.0, Sept. 1986.
27. J. D. Ullman, *Principles of Database Systems*, 2nd ed., Computer Science Press, Rockville, Md., 1983.

Received 15 August 1987