# Execution of Extended Multidatabase SQL

L. Suardi and M. Rusinkiewicz*

Department of Computer Science
University of Houston
Houston, TX 77204-3475

W. Litwin**

EECS Department
University of California
Berkeley, CA

## Abstract

*Multidatabase SQL (MSQL) is an extension of the SQL query language that provides new functions for non procedural manipulation of data in different and mutually non-integrated relational databases. We discuss the problems introduced by these new functions and analyze the semantics of multiple updates, global commitment and rollback. New language constructs are developed to allow declarative specification of multidatabase transactions. We also discuss the design and implementation of an environment for the execution of extended MSQL queries in a heterogeneous multidatabase environment.*

## 1 Introduction

In many application areas the information that may be of interest to a user is stored under the control of multiple database systems that were developed and are managed independently of each other. Although the existence of computer networks makes these databases easily accessible, the data in different databases are not integrated and the users must deal with multiple, autonomous and heterogeneous databases. The access to such databases is complicated by the possible naming, structure or type differences in the descriptions of the same real objects. Manipulation of data located in different databases requires functions that do not exist in currently used query languages such as SQL (Structured Query Language) [2].

The need to provide systematic access to multiple databases led to the emergence of the concept of *multidatabase systems* (MDBSs). A basic requirement of a multidatabase system is the existence of a *multidatabase manipulation language* [11] that facilitates access to data controlled by multiple Database Management Systems (DBMS). The databases that are members of some MDBS are called *Local Database Systems* (LDBSs). Multidatabases, also called Federated Databases [15], can be classified as *tightly coupled* or *loosely coupled*, depending on the level of autonomy of the LDBSs.

Multidatabase SQL (MSQL), an extension of the SQL query language, was proposed in [11] as an access language for loosely coupled multidatabase systems. Since SQL is a commonly accepted standard for relational databases, it is reasonable to think of MSQL as an emerging standard for multidatabase environments. The basic idea is that of providing SQL with new functions for nonprocedural manipulation of data in different and mutually non-integrated relational databases. Some MSQL capabilities, such as prefixing table names with database names, are already present in new SQL dialects, including SQL2.

The introduction of new functions for multidatabase manipulation in MSQL led to many open problems in the semantics and implementation of some operations. Some of these problems have emerged in our implementation of a subset of the language. These include the semantics of data definition, update, global commitment and rollback operations. In this paper these problems are analyzed and solutions are proposed as extensions to MSQL. We introduce new constructs for the specification of multidatabase transactions in MSQL and discuss the execution of MSQL queries in a heterogeneous multidatabase environment.

The paper is organized as follows: Section 2 is a brief overview of the main features of MSQL. Section 3 presents our critique of the language and the proposed extensions. Section 4 describes our MSQL implementation experience. Finally, Section 5 concludes the paper.

## 2 Language Concepts

MSQL provides a number of new features to facilitate the manipulation of multiple database systems. New schema manipulating operations include multidatabase creation and alteration; table definition in

*This work was supported in part by grants from Bellcoer and MCC and under the Texas Advanced Research Program, Grant number 3652008.

**Visiting EECS and HP Labs, Palo Alto



1063-6382/93 $03.00 © 1993 IEEE

multiple databases; importing of database and table schemas; creation and manipulation of multidatabase views and of virtual databases; and manipulation of multitables that are sets of tables originating in different databases. New data manipulation operations include joining of data that reside in different databases; retrieval of related data, with possibly different names and schemas, located in different databases; dynamic transformation of attributes' values; data transfer between databases; new built-in functions for aggregation and manipulation of multiple tables; and definition of interdatabase triggers. All these new functions can be expressed using powerful and concise MSQL statements.

To illustrate some of the features of MSQL, let us consider as an example two relational databases of car-rental companies, *avis* and *national*. Both databases contain information about cars that are either available or rented by customers (the schemas of both databases are presented in the Appendix). The two databases contain information about the same entity types and relationships of the real world. However, they present a number of heterogeneities in the names and structures used to model the same real entities. Let us suppose that we want to *retrieve from both databases the information about all cars with related code, type and daily rate (if it exists)*. The execution of such a query involves accessing both databases, generating partial results and merging them into the final result to be delivered to the user.

One problem that has to be solved in this example, for which standard SQL is not suitable, is the resolution of *naming heterogeneity*. Naming heterogeneity is caused by the use of different names to identify semantically equivalent objects in different databases (in our example, we have different table names: *cars* and *vehicles*). Another problem which cannot be solved by standard SQL is caused by *schema heterogeneity*, which exists when different database structures are used to model the same real objects. For example, in our query the column *rate* is present in the table *cars* but not in the table *vehicles*. The following single compact MSQL multiple query resolves both heterogeneities:

```
USE     avis national
LET     car.type.status BE cars.cartype.carst
                           vehicle.vty.vstat
SELECT  %code, type, ~rate
FROM    car
WHERE   status = 'available'
```

First, the query scope is defined by the *USE* statement. It contains the names of databases involved in the query. Then, naming heterogeneities are resolved by specifying the explicit semantic variable introduced by the *LET* statement. In the query body, references to the semantic variable *car.type.status* are substituted with the correct database objects names, specified in the *BE* clause. Another naming heterogeneity is resolved by the implicit semantic variable *%code*, which refers to both *code* and *vcode*. The wild character % stands for any sequence of zero or more characters and the proper names for the required column are derived automatically. Finally, schema heterogeneity is resolved by designating *rate* as an optional column, identified by the special character ~. The result of this multiple query is a *multitable*, which is a set of two tables. These two tables are generated as partial results by the accessed databases.

## 3 Extensions of MSQL

MSQL maintains the simplicity of relational languages for interactive use. The language deals with multiple objects in a transparent way and sets of subqueries are generated automatically whenever multiple identifiers are present in a global query. This makes MSQL very powerful. However, as stated in [11], MSQL functions led to many open problems at the implementation level. In this section, we propose solutions to some of these problems as extensions of the language.

### 3.1 Incorporate Services and Import Databases

Our prototype MDBS uses an *Auxiliary Directory* (AD) and a *Global Data Dictionary* (GDD). The Auxiliary Dictionary stores the information about the database systems (services) available in the multidatabase environment. For each service, we store the information needed to access the service, including its name, the address of the service site, the information about the access protocol and the information about the commit mode for the DML and DDL statements. The auxiliary directory can be updated by the INCORPORATE statement:

```
<incorporate statement> ::=
INCORPORATE SERVICE <service> [SITE <site>]
    CONNECTMODE  CONNECT | NOCONNECT
    COMMITMODE   COMMIT | NOCOMMIT
    CREATE       COMMIT | NOCOMMIT
    INSERT       COMMIT | NOCOMMIT
    DROP         COMMIT | NOCOMMIT
```

The statement inserts the information about a new database service (LDBMS) in the AD. The CONNECTMODE is CONNECT if the LDBMS supports

multiple databases, or NOCONNECT if the LDBMS supports one default database only. The COMMIT-MODE is COMMIT if the LDBMS works using automatic commit only and NOCOMMIT if it provides a two phase commit (2PCq) interface. The 2PC protocol is described in more detail for each DDL command such as CREATE, INSERT and DROP. This is necessary to cope with subtle heterogeneities that play an important role in the definition of the semantics of multidatabase commit and rollback.

The GDD is essentially a repository for the names of the database objects that are visible at the multidatabase level. It stores the names of tables together with the names, types and widths of their columns. This information is necessary to detect multiple identifiers in MSQL queries and to perform the substitution of implicit semantic variables. The GDD is manipulated by MSQL data definition statements such as CREATE DATABASE, CREATE TABLE, etc. It can also be built in a systematic way, starting from existing autonomous LDBSs. Once the service supporting a particular LDBS has been incorporated the names of its tables and columns can be imported from the Local Conceptual Schemas. This operation is performed by the IMPORT statement:

```
<import statement> ::=
 IMPORT DATABASE <database>
 FROM SERVICE <service>
   [ TABLE <table> [ COLUMN {<column>} ]] |
   [ VIEW <view> [ COLUMN {<column>} ]]
```

If the table name is not specified, the information about the structure of all tables designated as public, is imported. If the table name is specified, but column names are not, the whole table definition is imported. Finally, if column names are specified, partial table definitions can be imported. View definitions can be imported in a similar manner. The IMPORT operation replaces the definition of previously imported database objects, if necessary. It is assumed that database names are unique inside the federation, or that an aliasing mechanism exists that allows unique identification.

## 3.2 Vital Databases

In SQL there are three commands that modify the extension of a database: INSERT, DELETE and UPDATE. Their MSQL version allows the user to change the state of multiple databases simultaneously. Therefore, in the multidatabase environment the semantics of multiple updates must be carefully defined. The following example helps in understanding the problems involved in the implementation of such updates in a loosely coupled environment. Again using as an example the databases defined in the Appendix, let us assume that we want to *raise by 10% the fares of flights from 'Houston' to 'San Antonio' in Continental, Delta and United databases.*

```
USE     continental delta united
UPDATE  flight%
SET     rate% = rate% * 1.1
WHERE   sour% = 'Houston' AND
        dest% = 'San Antonio'
```

At the conceptual level the semantics of this multiple query are clear, but at the implementation level we have to consider several issues. The multiple query is decomposed into three subqueries that are executed by the three remote LDBMSs of databases Continental, Delta and United databases. Since the LDBMSs are autonomous and heterogeneous, they may use different 2PC protocols; some may not support 2PC. For some reasons (local conflicts, failure, deadlock, etc.) one or more LDBMSs may be forced to abort their local subqueries.

At the MSQL level the user has no way to control the execution of the subqueries, to specify dependencies among them, to specify alternative or compensating actions, etc. There is no definition of return value, of success or failure of a multiple query and, in particular, of multiple updates. The result of a multiple update may leave the multidatabase in a state that is inconsistent from the point of view of the global user. The only possibility to check if the multiple update was consistent would be to access each of the involved LDBSs and see what has happened.

### 3.2.1 Atomicity of MSQL queries

The concept of multiple query in MSQL is related to the concept of a *saga* [6]. The subqueries in a multiple query are related to each other and should be executed as a (possibly non-atomic) unit. A partial execution of the multiple query may be undesirable, and if it occurs, may need to be undone in some way, by the multidatabase system.

Our proposal is to allow the user to specify the desired level of consistency for the execution of a particular multiple query. This is obtained with an extension of the USE statement. Its syntax becomes:

```
<use statement> ::=
  USE [CURRENT][(] <(multi)database>
       [ <alias>)] [VITAL]
       [[(] <(multi)database> [<alias>)][VITAL]]
```

To illustrate this concept let us consider the example query presented in the previous section and assume that the updates to Continental and United databases must either both be completed successfully or none of them must be performed, while the update to Delta database is optional. The query corresponding to the above consistency requirements can be expressed as follows:

```
USE      continental VITAL delta united VITAL
UPDATE   flight%
SET      rate% = rate% * 1.1
WHERE    sour% = 'Houston' AND
         dest% = 'San Antonio'
```

The semantics of VITAL designators are similar to those defined in [5] for sub-sagas. Databases in the query scope are designated as VITAL or NON VITAL (default). If we assume that a multiple query is decomposed in such a way that there is at most one subquery per database, the VITAL designations are directly related to the generated subqueries. All VITAL subqueries are assumed to commit or abort, so that the desired degree of multidatabase consistency is maintained. A multiple query is successful when all VITAL subqueries commit. It aborts when all VITAL subqueries are rolled back. The execution is considered incorrect if some VITAL subqueries are committed and some other are not. All NON VITAL subqueries can be executed in autocommit mode and their results have no effect on the commitment or abort of the global multiple query. The set of VITAL databases is called *vital set*. Failure atomicity of a multidatabase query is enforced by the multidatabase system with respect to the vital set.

The user of a multidatabase language can specify different levels of consistency for the execution of MSQL queries, depending on the semantics of the operations s/he is going to perform. If all subqueries are NON VITAL the multiple query is always successful. On the other hand, when all databases are VITAL, we have traditional atomic transactions providing the "all or nothing" property. Different query evaluation plans are possible for the same multiple query, depending on the required level of consistency.

Assuming that all VITAL databases support 2PC, the above query can be executed as follows. First, the three database services are opened. Then, three tasks are submitted to execute the subqueries (local updates) remotely. Tasks that execute on databases designated as VITAL are not committed. If they are executed without errors their state becomes *prepared-to-commit*. Tasks executing on NON VITAL databases may be automatically committed and reach the *committed* state or rolled back and reach the *aborted* state. After the three tasks are executed the status of the VITAL tasks is checked. Depending on their status, appropriate actions are performed to assure global consistency. In section 4 we show how query evaluation plans, such as the one described above, can be expressed in the task specification language DOL [7].

### 3.2.2 Commit and rollback

Since MSQL is designed to access preexisting, autonomous and heterogeneous databases, the semantics of commit and rollback operations depend on the commitment protocols provided by those database systems. LDBMSs supporting automatic commit and LDBMSs supporting user-controlled 2PC may be involved in the same query. LDBMSs which support 2PC may adopt different protocols. For example, in our implementation both Ingres and Oracle provide 2PC, but with different protocols. One of the DBMSs allows DDL commands to be rolled back while another automatically commits them together with all previously issued uncommitted statements. These possible heterogeneities must be captured at the multidatabase level, if we want to define clear semantics for global commit and rollback.

If a multiple query performs operations for which all accessed LDBMSs provide the same 2PC protocol the implementation of global commit is straightforward. Assuming that the LDBMSs have a *visible prepared-to-commit state* an evaluation plan can be specified which permits all subqueries to be committed or aborted. A subquery enters its prepared-to-commit state when it completes the execution of its operations and leaves this state when it is committed or rolled back. However, if a multiple query accesses LDBMSs providing different 2PC protocols, things are more complicated. For example, if all the accessed LDBMSs automatically commit every database operation, there is no way to define clear semantics for the global commit and rollback statements. In this case, the only way to simulate rollback operations is to use user-defined compensating actions (section 3.3). More problems arise if some of the accessed LDBMSs use automatic commit and some others use 2PC.

Query evaluation plans that are generated by the multidatabase system must deal with all the described heterogeneities and produce consistent results. The VITAL designators introduced above can be used to provide a declarative mechanism to clarify the semantics of global commit and rollback operations.

All MSQL commit and rollback operations refer to databases that are VITAL in the current scope. The evaluation plan can be structured so that all VITAL

subtransactions are successfully completed or rolled back (or compensated). The evaluation plan will contain synchronization points whenever explicit commit or rollback operations are issued, the current query scope is changed, or the last MSQL statement is terminated. If all VITAL databases are either prepared or committed at the synchronization point, the subqueries that are in the prepared state will be committed. Otherwise all VITAL subqueries will be rolled back (or compensated). MSQL users can decide which databases should participate in global commit and rollback operations by means of the VITAL designators.

### 3.3 Compensation

As we have mentioned in the previous section, the described semantics of the VITAL designators are not applicable if the user wants to include in the vital set databases that do not support 2PC. In this case our prototype MDBS raises an error condition and refuses to process the query. This is because if two or more of such databases are VITAL, it is not possible to enforce failure atomicity with respect to the vital set. Nothing can be done if one of them commits and another one aborts the related subquery, and the global consistency is violated.

A possible solution to this problem is the use of *compensation* [6, 8]. Compensating actions can semantically undo the effects of committed subqueries, although they do not necessarily return the database to the state that existed when the execution of the subqueries began. The original MSQL specifications do not provide mechanisms for compensation of particular subqueries of a multiple query. We propose to extend the standard MSQL manipulation statements by allowing a clause to explicitly specify compensating actions:

```
<manipulation statement> ::=
  <USE statement> <LET statement>
  <SELECT stmt> | <INSERT stmt> |
  <UPDATE stmt>
  <COMP statement>

<COMP statement> ::=
  COMP  <database name | alias>
        <compensating subquery>
```

For each VITAL database in the scope of the query that does not support 2PC, the user must provide a COMP clause in which the needed compensating actions are specified. For example, assuming that the Continental database does not provide 2PC, the previous multiple update can be rewritten in the following way:

```
USE     continental VITAL delta united VITAL
UPDATE  flight%
SET     rate% = rate% * 1.1
WHERE   sour% = 'Houston' AND
        dest% = 'San Antonio'
COMP    continental
        UPDATE flights
        SET    rate = rate / 1.1
        WHERE  source = 'Houston' AND
               destination = 'San Antonio'
```

With the specification of the compensating action for the local update to database Continental the original semantics of the VITAL designator are preserved. If the Continental update is aborted, the United update can be rolled back. If the United update is aborted, the Continental update can be compensated. The evaluation plan for such a multiple query could be generated as follows.

First, the tasks corresponding to the three subqueries are executed as in the example of section 3.2. The second part is more complicated and considers all possible execution paths. If the Continental database has committed and the United database is in the prepared-to-commit state, the United subtransaction can be committed and the MSQL query is successful. If Continental has committed and United has rolled back (aborted), Continental must be compensated and the MSQL query is successfully aborted. If Continental has aborted and United is in the prepared-to-commit state, United must be rolled back and the MSQL query is successfully aborted. Finally, if both databases have aborted the MSQL query is successfully aborted. The Delta database, being NON VITAL, does not affect the success or failure of the MSQL query.

The introduction of VITAL designators and compensation is a step in the direction of the specification of multidatabase transactions in relational environments. MSQL queries which specify VITAL subqueries and eventual compensating actions can be considered as small transactional units. The natural next step is the specification of more complex transactions.

### 3.4 Specification of Multidatabase Transactions

MSQL inherits from SQL its limited ability to specify transactions. This status of the most widely used database access language does not reflect the current status of the research in the area of transaction management [6, 10, 13, 3, 14, 4]. However, the literature on transaction specification is much more limited [9, 7, 1]. The ability to specify and execute transactions with various semantics is highly desirable, especially in

a multidatabase environment. We propose an extension of MSQL to incorporate some elements of *Flexible Transactions* [3].

Flexible transactions may provide *Function Replication*, which exists when a given task can be accomplished on different databases. For example, if multiple car rental databases are available to the multidatabase user, then the rent-a-car task can be performed on any of those databases. MSQL can naturally take advantage of this situation by specifying multiple queries that perform functionally equivalent tasks on different databases in a transparent way.

*Compensation* is useful when we want to relax some of the Atomicity, Consistency, Isolation and Durability (ACID) properties of traditional transaction models or when we want to cope with various kinds of heterogeneities. This is the case in the multidatabase environment, where transactions are potentially long lived and the enforcement of full isolation or atomicity may be potentially expensive. Compensating transactions can semantically undo the effects of committed transactions. This allows the global transaction to reveal its partial results by committing some of the subtransactions that may need to be compensated later. The isolation granularity of the global transaction is reduced to the subtransaction level. The performance of the system may be improved through earlier release of the resources held by global transactions.

*Acceptable termination states* are used to relax the failure atomicity requirements of global transactions. An acceptable termination state is an execution state in which the global transaction achieves its objectives. Frequently, there is more than one acceptable state for a global transaction.

In the following, we describe extensions of MSQL that allow the user to specify multidatabase transactions consisting of two or more MSQL queries. These extensions capture the described concepts of function replication, compensation and relaxed failure atomicity[1].

An *MSQL multitransaction* is a set of MSQL queries. It is started by the BEGIN MULTITRANSACTION statement and terminated by an END MULTITRANSACTION statement. Among the traditional ACID properties isolation is not supported, since the partial results of committed subqueries may become visible to concurrent transactions before the commitment of the multitransaction. The multitransaction is specified as follows:

[1]The Flexible Transaction Model provides a number of additional features, including an extensive set of execution dependencies that may be specified among subtransactions of a flexible transaction. These execution dependencies are not modeled in the extended MSQL.

```
<MSQL transaction specification> ::=
  BEGIN MULTITRANSACTION
        <list of queries>
        <commit statement>
  END MULTITRANSACTION
<commit statement> ::=
  COMMIT <list of acceptable states>
<list of acceptable states> ::=
  <db | alias> AND ... AND <db | alias>
```

The COMMIT statement specifies the acceptable termination states for a multitransaction. An acceptable state is expressed as the conjunction of the subqueries whose success is required for the success of the global transaction. Subqueries that are not specified in one acceptable state are assumed to be aborted or compensated. We refer to the subqueries of a multitransaction using the database names. Therefore, one acceptable state becomes a conjunction of database names or aliases. This is possible because MSQL queries are assumed to generate at most one subquery per database. The aliasing mechanism in the USE statement allows database names to be unique inside a multitransaction specification. The conditions for success of a multitransaction are expressed as an implicit disjunction of acceptable termination states. The order in which acceptable states are specified may indicate the user preference.

As an example let us consider a travel agent who needs to prepare a trip plan for a customer [3]. Let us assume that the flight reservations may be made with either Continental or Delta, and that a car may be rented from either Avis or National. The trip planning is successful if a flight is reserved with one of the two airlines and a car is reserved with one of the rental companies. The preferred solution is to fly with Continental and to drive a National car, an acceptable alternative is to fly with Delta and to drive an Avis car. The other flight/car combinations are undesirable. An MSQL multitransaction that satisfies the above requirements can be specified as follows:

```
BEGIN MULTITRANSACTION
 USE      continental delta
 LET      fltab.snu.sstat.clname BE
          f838.seatnu.seatstatus.clientname
          f747.snu.sstat.passname
 UPDATE   fltab
 SET      sstat = 'TAKEN', clname = 'wenders'
 WHERE    snu = (  SELECT  MIN(snu)
                   FROM    fltab
                   WHERE   sstat = 'FREE');
 USE      avis national
 LET      cartab.ccode.cstat BE
```

```
        cars.code.carst
        vehicle.vcode.vstat
UPDATE  cartab
SET     cstat = 'TAKEN', from = '07-04-64',
        to = '04-16-92', client = 'wenders'
WHERE   ccode = ( SELECT  MIN(ccode)
                  FROM    cartab
                  WHERE   cstat = 'FREE');

COMMIT
        continental AND national
        delta AND avis
END MULTITRANSACTION
```

Four different databases are accessed by this multitransaction. The first MSQL multiple query performs the flight reservation on both Continental and Delta. This is accomplished by setting the value of the column *sstat* to TAKEN, for the FREE seat with the lowest number. The second multiple query performs the car reservation on both Avis and National in a similar way. This is an attempt to take advantage of function replication.

Since it is not desirable to have two flight reservations and two cars, some exclusion constraints must be specified as conditions for successful termination. This is accomplished specifying two acceptable states in the COMMIT statement. For example, the acceptable state *continental AND national* means that the multitransaction is successfully committed when the subtransactions on Continental and National are committed, while the subtransactions on Delta and Avis are rolled back or compensated. The exclusion of Delta and Avis subtransactions is implicit in the specification, that is equivalent to *continental AND national AND NOT delta AND NOT avis*. In addition, an implicit OR is assumed between the acceptable states.

The acceptable states will be checked in the order in which they are specified, after the execution of the two multiple queries. The first acceptable state that can be reached from the execution state of the four subqueries will be the final state produced by the multitransaction. If neither of the acceptable states can be reached the multitransaction fails and all subqueries will be rolled back or compensated.

If all databases accessed by the multitransaction support 2PC, these subqueries are kept in the prepared state until the COMMIT point is reached. Then, the execution states reached by the subqueries are compared to the acceptable states and some of the subqueries are committed while other are rolled back. If some of the accessed databases do not support 2PC, compensation must be specified for all subqueries that are executed on those databases. The evaluation plan specification will be more complicated in this case since it will have to consider all the possible execution paths to decide when compensation is necessary.

# 4 Execution of Extended MSQL

In this section we discuss the implementation of the execution environment for the extended MSQL. The implementation is described in detail in [16]. In our implementation, we used the Narada distributed execution environment that was developed at the University of Houston. Narada provides support for transaction and communication services and uses a task specification language, DOL, to describe the execution of distributed applications in a heterogeneous computing environment [7].

## 4.1 System Architecture - Narada Environment

The main components of the system for the execution of Extended MSQL are the *translator* of MSQL queries into DOL programs, the *engine* for the execution of DOL programs in the Narada Environment and *local access managers* (LAMs) for the transparent access of remote databases and other information resources (Figure 1).

All these components cooperate by exchanging various information. The translator receives MSQL queries and produces DOL programs that are passed to the engine. The engine executes these programs and coordinates the actions of various LAMs, exchanging messages, data and command files with them. LAMs execute local commands and produce partial results, which are sent either to the engine or to other LAMs. Finally, the partial results are collected in one database, acting as the *coordinator*, and the final results are produced. The translator receives back DOL return codes, which describe the execution status reached by the engine. These codes are used as MSQL return codes. The Narada resource directory contains up-to-date information about all the services known to the DOL engine. The information includes physical addresses, communication protocols, login information and the data transfer methods used for all nodes in the multi-system environment.

Narada is an environment for the specification and execution of multisystem application and can be used for many purposes. Here it is tailored to the execution of multidatabase queries specified in MSQL. DOL is used to specify such applications, without dealing explicitly with the details of task synchronization and data exchange. Once an application is defined, it can

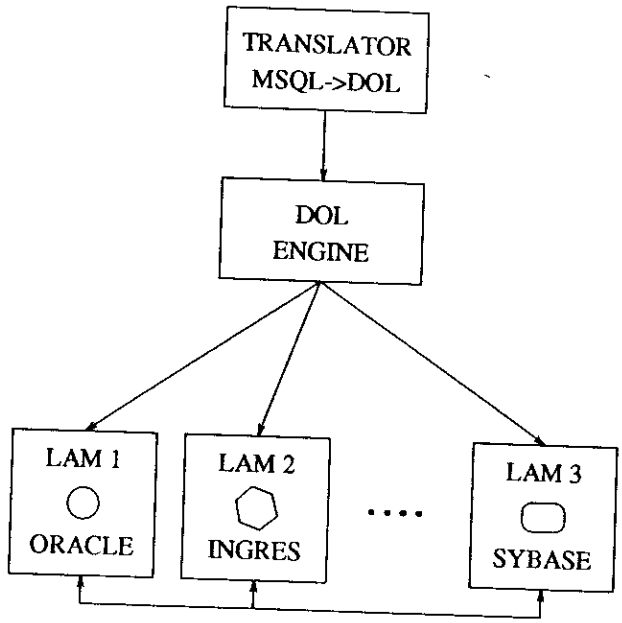


**Figure 1: System Components**

be executed automatically, providing location transparency and resolving most of the hardware and software incompatibilities. MSQL queries are automatically mapped into DOL programs.

Narada provides transparent access to multiple existing systems (databases, knowledge bases, software packages, etc.). Since none of these systems is modified, they retain their autonomy. Narada allows the user to execute a multi-system application by specifying different actions, their logical dependencies, data paths among them, and the possible concurrency. To allow this, the DOL language provides primitives for task invocation at both local and remote sites, task synchronization, data flow control and conditional execution handling. Although DOL can be used to specify multi-system applications, it is neither a multidatabase query language nor a transaction specification language. DOL may serve as an intermediate language, which can be automatically produced starting from higher level specifications of multidatabase queries and transactions.

## 4.2 Schema Architecture

The multidatabase schema architecture used in our implementation is presented in Figure 2. We assume that all LDBSs have a relational interface and can be accessed through their Local Conceptual Schema (LCS). The public tables belonging to LCSs can be exported to the multidatabase level. At the multidatabase level there are a Global Data Dictionary (GDD) and an Auxiliary Dictionary (AD). The system is organized as a federation. LDBMSs may be incorporated in the federation and their description stored in the auxiliary dictionary. When a new LDBMS is incorporated its capabilities must become known to the federation and stored in the auxiliary dictionary. These capabilities range from the possibility to support more than one database to the adopted one-phase or two-phase commit (2PC) protocol.

Then, the conceptual schema of incorporated databases can be imported and stored in the global dictionary. Since schema translation and integration are not used, the model can be classified as a loosely coupled federated database system [15].

## 4.3 MSQL Query Processing

The execution of MSQL queries is divided into the following phases: multiple identifier substitution, disambiguation, decomposition, execution plan generation and execution. The interpreter processes each MSQL query as follows. If the query is multiple, all possible substitutions of multiple identifiers are generated and non pertinent queries are discarded during disambiguation. Then, each global fully qualified elementary query Q is decomposed into SQL subqueries $q_1, \ldots, q_n$ and a global modified query Q'. The decomposition of Q is based on the location of the accessed data items and is performed using query graph analysis [12]. The global query is transformed into a set of the largest possible local subqueries, one for each involved LDBS.

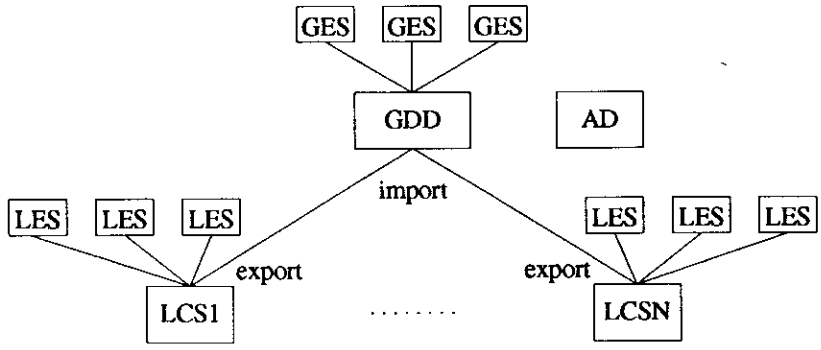


**Figure 2: Schema Architecture**

One of the LDBSs is designated as the *coordinator* and will evaluate the modified global query.

The subqueries and the modified global query are then incorporated in a DOL execution plan. The execution plan can specify exception handling, 2PC operations, compensating actions, parallel execution, etc. An MSQL query can be mapped into a number of different DOL plans. The ability to generate optimal plans depends on the extent of the information stored in the GDD and AD. Due to the loosely coupled nature of the MSQL approach the optimization is likely to be related more to data flow control and parallelism than to database operations.

For example, the query specified in section 3.2 is mapped into the following DOL program:

```
DOLBEGIN
    OPEN continental AT site1 AS cont;
    OPEN delta AT site2 AS delta;
    OPEN united AT site3 AS unit;
    TASK T1 NOCOMMIT FOR cont
        { update for continental }
    ENDTASK;
    TASK T2 FOR delta
        { update for delta }
    ENDTASK;
    TASK T3 NOCOMMIT FOR unit
        { update for united }
    ENDTASK;
    IF (T1=P) AND (T3=P) THEN
        BEGIN
            COMMIT T1, T3;
            DOLSTATUS=0; { return code }
        END;
    ELSE
        BEGIN
            ABORT T1, T3;
            DOLSTATUS=1; { return code }
        END;
    CLOSE cont delta unit;
DOLEND
```

The DOL OPEN statement connects to a known service at the specified site and establishes a reliable communication channel to it. The connection protocol and the data exchange protocol are specified in the Narada service directory and therefore are completely transparent to the MSQL user [2]. The TASK statement is used to pass the local commands to be executed by the service. In our case, these commands are SQL statements for the local database systems. After the three tasks are executed, the status of VITAL tasks is checked. The tasks are executed in a NOCOMMIT mode; if successful their status becomes prepared-to-commit (P). Depending on the execution status of the tasks, appropriate actions are performed to commit or abort the transaction.

If Continental database does not provide the prepared to commit state and a compensating transaction is provided, we have two different ways to abort the MSQL query at the global level. One way is to compensate on database Continental, when database United has rolled back its subquery. Another way is to rollback on database United, when database Continental aborted its subquery.

In a similar fashion, DOL programs are constructed to execute the multitransactions shown in section 3.4.

## 5 Conclusions

In this paper we described the first implementation of a subset of MSQL. The experience has shown that the proposed functions led to many open problems at the levels of the implementation and of the semantics of some operations. These problems are mainly concerned with non-retrieval operations such as data definition, updates, global commit and rollback. In section 3 these problems are analyzed and solutions are proposed as extensions of MSQL. Since MSQL inherits from SQL the very limited ability to specify transactions, extensions of MSQL have been proposed to incorporate some

[2] TCP/IP and an ISODE prototype are used by the current implementation.

elements of Flexible Transactions. The ability to specify and execute transactions with various semantics is very important in a multidatabase environment.

The work described here can be continued and expanded in many directions. Since DOL allows the specification of flexible transactions, Extended MSQL transactions can be easily mapped into DOL programs and executed. The resulting DOL programs may also be optimized. This seems to be a promising approach to the optimization of global queries without the use of a global schema. The optimization will be related more to data flow control and parallelism in execution of queries at different sites than to individual database operations.

## A Appendix: Example Database Schemas

Database CONTINENTAL (airline)
flights (flnu, source, dep, destination, arr, day, rate)
f838 (seatnu, seatty, seatstatus, clientname)

Database DELTA (airline)
flight (fnu, source, dest, dep, arr, day, rate)
fnu747 (snu, sty, sstat, passname)

Database UNITED (airline)
flight (fn, sour, dest, depa, arri, day, rates)
fn727 (sn, st, sst, pasna)

Database AVIS (car rental)
cars (code, cartype, rate, carst, from, to, client)

Database NATIONAL (car rental)
vehicle (vcode, vty, vstat, from, to, client)

## References


[1] O. Bukhres, Jiansen Chen, Jindong Chen, A. Elmagarmid, Y. Leu, and G. Zhu. IPL: The InterBase Parallel Language. In *Second International Workshop on Research Issues on Data Engineering: Transaction and Query Processing*, February 1992. Tempe, Arizona.

[2] C.J. Date. *A Guide to The SQL Standard.* Addison-Wesley Publishing Company, 1987.

[3] A.K. Elmagarmid, Y. Leu, M. Rusinkiewicz and W. Litwin. A Multidatabase Transaction Model for InterBase. In *Proceedings of the 16th International Conference on VLDB*, 1990.

[4] A. Elmagarmid, editor. *Database Transaction Models.* Morgan-Kaufmann, San Mateo, CA, 1992.

[5] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem. Coordinating multi-transaction activities. Technical Report CS-TR-247-90, Princeton University, February 1990.

[6] H. Garcia-Molina and K. Salem. SAGAS. In *Proceedings of ACM SIGMOD Conference on Management of Data*, 1987.

[7] Y. Halabi, M. Ansari, R. Batra, W. Jin, G. Karabatis P. Krychniak, M. Rusinkiewicz, and L. Suardi. Narada: An Environment for Specification and Execution of Multi-System Applications. In *Proceedings of the Second International Conference on Systems Integration*, 1992.

[8] H. F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *Proceedings of the 16th International Conference on VLDB*, 1990.

[9] E. Kuehn, F. Puntigam, and A. Elmagarmid. Transaction Specification in Multidatabase Systems Based on Parallel Logic Programming. *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, April, 1991.

[10] J. Klein and A. Reuter. Migrating Transactions. In *Future Trends in Distributed Computing Systems in the 90's*, Hong Kong, 1988.

[11] W. Litwin. MSQL: A Multidatabase Language. *Information Sciences*, 1990.

[12] M. Rusinkiewicz and B. Czejdo. An approach to query processing in federated database systems. In *Proceedings of the Twentieth Hawaii International Conference on System Sciences*, 1987.

[13] A. Reuter. Contracts: A means for extending control beyond transaction boundaries. In *Presentation at Third International Workshop on High Performance Systems*, September 1989.

[14] M. Rusinkiewicz and A. Sheth. Polytransactions for Managing Interdependent Data. *Data Engineering Bulletin*, 14(1), March 1991.

[15] A. Sheth and J. Larson. Federated database systems for managing distributed, heterogeneous and autonomous databases. *ACM Computing Surveys, Vol.22, No.3*, September 1990.

[16] Luigi Suardi. Execution of Extended MSQL. Master's thesis, Department of Computer Science, University of Houston, June 1992.