FEDERATED DATABASE MANAGEMENT SYSTEMS: AN ARCHITECTURE TO DISTRIBUTE PERSISTENT DATA THROUGH HETEROGENEOUS DATABASES

Thierry MILLAN, Myriam LAMOLLE, Frédéric MULATERO
IRIT - CNRS (UMR 5055)
Université Paul Sabatier
118, route de Narbonne
31062 Toulouse Cedex
France
E-mail: (millan, lamolle)@irit.fr

KEYWORDS
interoperability, Federated Database Management Systems

ABSTRACT

This paper sets out the general architecture of the Federated DataBase Management Systems (FDBMS). The aim of this architecture is to ensure that integrating new DataBase Management Systems (DBMS) in the FDBMS does not disturb the pre-existing applications. So, this federation preserves the autonomy of each database: pre-existing applications, concurrency control, garbage collector, etc. This architecture uses three visibility levels of objects (local, shared local and global) according to the applications (local and global). Each site requires three modules (client, server, global garbage collector) to manage shared local and global objects. In addition, each site keeps on with the management of the local objects handled by their pre-existing applications.

INTRODUCTION

There is a new trend to federate pre-existing and independent DBMS into Multi-DataBases Systems (MDBS). This study aims at providing access to data stored in databases. To minimise the modification of each database management systems, the following constraints must be taken into account. Firstly, data must be accessible from pre-existing applications. Secondly, usual users must not be disturbed by the integration of the pre-existing applications. Finally, the emergence of global applications should not disturb the pre-existing local application.

Remark: Since databases that constitute the federations can be either relational database, or relational-object databases or object-oriented databases, we will consider data as objects in this paper.

APPLICATIONS AND OBJECTS

Two types of applications co-exist in a FDBMS. Local applications are limited to the manipulation of local objects stored in one DBMS. These manipulations escape the control of the MDBS. Global applications gain transparent access to shared local data located in the different DBMS that are federated by the MDBS.
For each DBMS involved in the federation, objects are partitioned in three disjointed sets.

**Local objects** are only accessed by local applications. References between local objects are local to one DBMS. Local objects constitute the first partition. **Shared local objects** are accessed both by local and global applications. Such objects are local objects that are made visible to the global applications. They do not contain external references. Thus, local applications that are not under the control of the MDBS cannot reach a global object through a shared local object. Shared local objects constitute the second partition.

**Global objects** are shared by global applications that are under the control of the MDBS. A global object may contain external references that are references to global objects located in a remote DBMS. Global objects constitute the third partition.

With regard to referential persistency (Atkinson et all 1983) objects are said to be persistent if they are named or referenced by a named object. In order to preserve DBMS autonomy, each DBMS manages its own set of local names for persistency roots of local objects. As mentioned above, two types of reference co-exist in the FMDBS: *local references* and *external references*. Local references are used to identify or send operations to local and shared local objects inside a local DBMS. External references are necessary to access global and shared local objects in the FMDBS context. While local references provide direct access to objects inside a DBMS, external reference provides indirect access to objects across the network. As in (Lang 1983), an external reference to object \( o \) is composed of a local reference to an exit item stored in the local DBMS (see Figure 1), which in turn references an entry item stored in the DBMS containing object \( o \), which itself contains a local reference to object \( o \). The reference between an entry item and an exit item is a remote reference that can be implemented as the concatenation of a DBMS identifier and a local reference.

**GENERAL ARCHITECTURE**

There is no restriction in combining global objects in order to construct complex objects. Figure 1 shows an example where global object \( o1 \) stored in DBMS1 is composed of global object \( o2 \) stored in DBMS2, that in turn is composed of object \( o3 \) stored in DBMS3, that in turn is composed of object \( o4 \) stored in DBMS1.

The figure 2 shows that there is no limitation in the length of the composition string of complex objects and that there is no hierarchy between the DBMS involved in the composition. As a consequence, the same DBMS can in turn fulfill the role of a client or of a server. It is considered as a client when it calls the interface of a global object stored in a remote DBMS, while it is considered as a server when it implements the interface of a global object. In order to integrate a new DBMS into the federation, three modules have to be developed, namely the DBMS client module, the DBMS server module and the DBMS global garbage collector module (see figure 2).

**Different modules**

**DBMS client module**

The client module is in charge of providing both the MDBS interface to global applications and the global object interface. The MDBS interface is composed of the following operations: connection to the FMDBS, disconnection, commit a transaction and abort a transaction. The connect operation initiates a first transaction for the global application and the disconnect operation commits the last transaction of the application. The commit operation must ensure the durability of all the operations performed by the transaction (Atomicity and Durability) and must follow a two step commit protocol since the MDBS interface is in a distributed context. The abort operation must ensure that all the operations of a transaction are cancelled (Atomicity).
The global object interface consists in sending global object method calls to the appropriate DBMS server module as well as to provide access to the list of global names. This can be done by means of proxy following by CORBA, whose objective is to encapsulate object method calls through exit items. It is important to mention here that since we are in a transactional context, each global object operation is performed for one transaction, the identifier of which is transmitted either by the transaction or by other means.

Exit items are managed by the client module. Exit items are implemented as a class of local objects stored in the DBMS which in turn is associated to the client module so that they can be referenced through a local reference. If the exit item is already assigned to a global object, the client item creates a new exit item so that there is one exit item for each external reference held in the DBMS; this is the case even if the exist items refer to the same object. Having several exit items pointing to the same object avoids contention on exit items resulting from the DBMS transaction management mechanism (an exit item updated by one transaction is locked for the whole transaction).

DBMS server module
On the server side, the server module provides the mirror interfaces of the client module interfaces. These interfaces provide access to the transactional facilities mentioned above and, through an entry item, transmit global objects methods calls to the appropriate object stored in a local DBMS. This can be done by mean of stubs like in CORBA. Entry items are managed by the server module. Contrary to exit items, entry items are not stored in the local DBMS. In this way, it is easier to provide uniform remote references in each server module. Another advantage of implementing entry items outside DBMS is to ensure that coherent parallel updates can be performed on entry items by means of latches (Mohan et all 1989) and instead of transactions. Latches are instant duration locks that guarantee the coherency property of parallel updates without providing the atomicity property of transactions.

Garbage Collection (Mulatéro et all. 1998a) (Mulatéro et all. 1998b)
The purpose of the global Garbage Collector (GC) is to make sure that entry items, global object cells and exit items are destroyed when they are no longer to be used. On the other side, the purpose of a local GC is to collect the local objects no longer used in a DBMS. The global GC is a distributed task and is implemented by modules associated with each DBMS. Each DBMS component is supposed to have its own local GC and no assumption is made with regard to the behaviour of the local GC. The global GC combines the use of inverse reference lists with a reverse mark and sweep algorithm in order to solve both the dead object cycles and the propagation problems. It has the following properties. Firstly, there is no interaction between the global GC and the local GC. Each local GC continues its execution without

![Figure 2: General Architecture](image-url)
taking account of the execution of the global GC. Secondly, the global GC is incremental. It runs in concurrence with the applications already using the databases. Thirdly, the global GC does not need to stop the application from running. It is able to collect objects without accessing the whole databases. Fourthly, the Global GC requires few interactions with transactions. The transactions have a higher priority than the global GC. Fifthly, the global GC is able to detect dead object cycles. Sixthly, global synchronisation of DBMS sites is not required. There is no constraint concerning the execution of each site. Finally, the Global GC works exclusively on entry and exit items without accessing global object cells stored in DBMS. Consequently it implies few I/O overhead for the DBMS.

**About Transaction Management**

The specificity of database objects is that they have to respect the ACID properties (Gray and Reuter 1993), namely: Atomicity, Coherency, Isolation and Durability. Global objects, as well as local objects, must satisfy ACID properties. Consequently, database objects are manipulated through transactions. The MDBS distinguishes global transactions accessing global and shared local objects from local transactions executed on each DBMS. A global transaction is supported by one local transaction per DBMS involved in the global transaction. As a global transaction commits, the commitment of the involved local transactions is synchronised though a two phase commit protocol (Gray and Reuter 1993). This interface is encapsulated by the DBMS client and server modules.

**CONCLUSION**

In this paper, we show how to federate existing DBMS without any modification of the local applications. We consider data as objects with three visibility levels according to their use (local, shared local and global objects). The MDBS satisfies the constraints expressed in the introduction. This architecture is a solution to federate database management systems that presents the best cost-effectiveness ratio thanks to the re-use of the pre-existing databases, software and hardware, and the minimisation of the training times.

Currently, we are carrying out two studies. The first one concerns the development of persistent application using heterogeneous languages (Ada, Java). Such bindings could offer valuable solutions to develop global applications that exploit global objects (Internet applications using Java for example). Furthermore, they could also make easier the migration of local applications to global applications. The second one is the optimising of the garbage collector module. The aim of this study is to propose an incremental garbage collector concurrently to transactions. The main objective is to not stop the transactions during reclaiming and reallocating of the free memory space; so this GC processes in background. In addition, we optimise the GC using an algorithm based on the graph theory (Bergé 1973).

**REFERENCES**


Lang B., Queinnec C., Piquer J. 1992. “Garbage Collecting the World” in proceedings of the Nineteenth Annual ACM Symposium of Principles of Programming Languages

