Active Object-Relational Mediators

Thomas Kudrass, Andreas Loew, Alejandro P. Buchmann
TH Darmstadt, FB Informatik, FG Datenverwaltungssysteme 1
D-64293 Darmstadt, Frankfurter Str. 69 A
E-Mail: {kudrass, loew, buchmann}@dvs1.informatik.th-darmstadt.de

Abstract

This paper describes an active object-oriented mediator for the enforcement of global consistency between relational legacy databases. We discuss the problem of integrating several local relational systems into a federated system by the usage of an object-oriented mediator system. We explore how relational DBMSs can be enhanced to signal local updates that may violate global constraints without sacrificing too much autonomy and present a database gateway for detection, logging and signalling. We show how the gateway is embedded into the architecture of an object-relational mediator system. We give a solution to the problem of mapping SQL commands to method calls in a C++ based system using a so-called mediator generator. Furthermore, we discuss how the federated system can be enriched by rule mechanisms that make the mediator behave actively.

1 Introduction

Modern information systems tend to integrate existing heterogeneous systems into federations while preserving the autonomy of the participating subsystems. This guarantees that existing applications can run without changes of their code and new applications can be developed that access the federated data. This approach is a practical way to incorporate legacy applications into future information systems.

When integrating heterogeneous systems, the problem of global consistency maintenance spanning multiple databases has to be solved by the federated system. There is some work dealing with interdatabase dependencies [22, 6, 17] resulting from the integration of overlapping or dependent data that has been maintained autonomously. Besides the interdependencies, all other kinds of integrity constraints as described in [1, 8] for an object-oriented model (like uniqueness, referential integrity, structural constraints) can be applied to a multidatabase environment. In this paper, we focus on local relational databases, which have become the most popular platform in the last ten years, but will be the legacy data of the future. We prefer an object-oriented canonical data model due to its semantic richness and openess against possible future object-oriented local systems.

Active mechanisms are a promising approach for consistency maintenance in federated systems. Although the usage of the active object paradigm was already proposed in the context of the DOM project [2], existing prototypes of active OODBMS (e.g. REACH [4], SAMOS [12], ODE [13], NAOS [10], Sentinel [9], Chimera [7]) simply enhance homogeneous database systems with active mechanisms. Analyzing the limitations of existing active DBMSs [3, 25], it has to be explored to what extent active features can be applied in federated systems [21].

Due to the coexistence of several data models and the local autonomy, some of the principles of homogeneous active DBMSs cannot easily be transferred to a federated system. One of the crucial issues is the detection of events both at the local level (i.e. a local application interface) and at the global level that is available to users of the federation. The problem of local event detection derives from the fact that legacy systems were neither designed nor engineered to signal the relevant events. Figure 1 sketches a multidatabase environment comprising different local databases. The database may be updated at two access levels. The federated component responsible for consistency maintenance has to be aware of events caused by global users as well as local users accessing single databases through their existing local interfaces.

One possible solution we are presenting here is based on the integration and mapping of local database operations to operations in the federated system, where they can be detected in the same way as globally submitted statements. Beyond the mapping from C++ operations to SQL statements, which has to be done

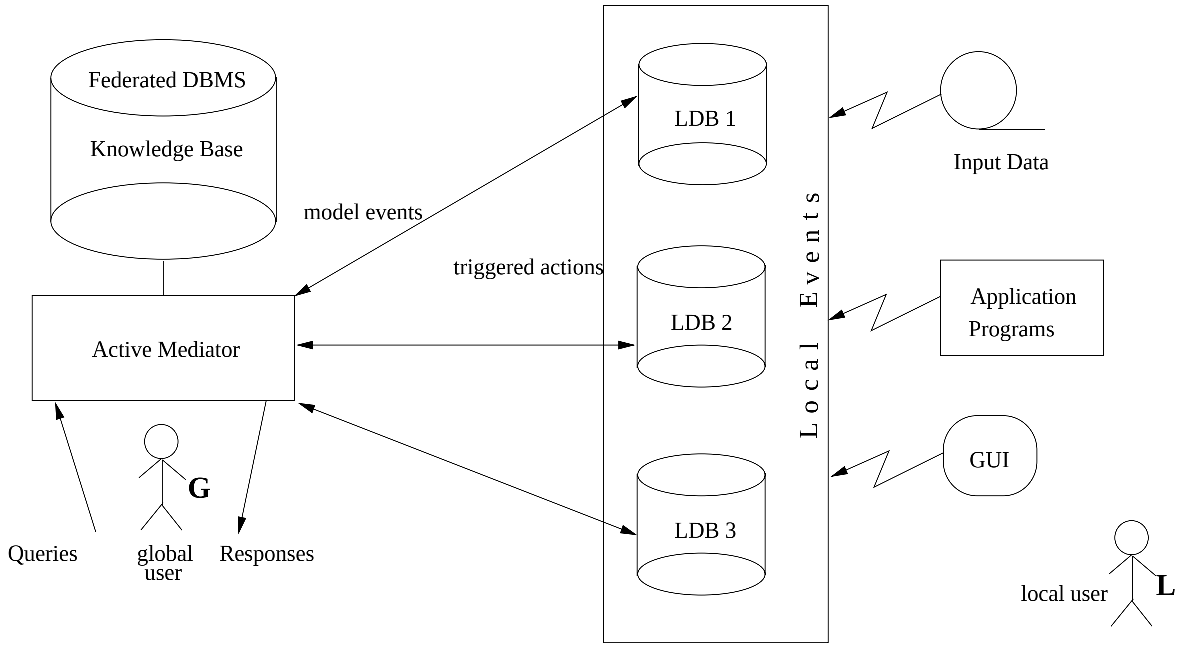

Figure 1. Multidatabase Environment (Sample Scenario)

by the federated system, we present a solution that reversely maps local SQL statements to method calls. These method calls can then trigger the rules that restore consistency.

The paper is organized as follows: In section 2, we describe our gateway approach to detect local database events. Section 3 discusses the structural mapping between tables and classes and introduces the chosen mediator platform Persistence[1], an object-relational system. In section 4, the behavioral mapping of SQL statements to C++ method calls we have contributed to Persistence is presented. In section 5, the architecture of our mediator prototype is sketched. We describe the concept of a mediator generator in section 6. Section 7 is dedicated to active extensions of the mediators we are working on. The paper concludes with an evaluation of the presented approach and gives an outlook on open issues under research.

2 Monitoring Local Systems - The Gateway Approach

2.1 Local Database Event Detection

When integrating a relational DBMS into a federated system, it has to be identified which SQL commands submitted at the local application interface are the significant ones that have to be monitored. Table 1 gives an overview of those database events to be monitored by local components with respect to global consistency.

Local event detection can be implemented through various mechanisms (for a classification of wrapping see [27]):

- trigger mechanisms
- application wrapping
- database wrapping

Local application systems can be made active e.g. by exploiting *trigger mechanisms* of the modern relational DBMSs, but they show too many restrictions, which make their usage impractical for the purpose of detection and signalling. The main drawback is the restriction of triggers to react on INSERT, UPDATE and DELETE commands, and their limited number per command and table, which may cause conflicts with existing triggers in local schemas. In Sybase, for example, for a given table it is possible to have only one trigger each on INSERT, UPDATE and DELETE commands, so when the legacy application to be integrated into a federation happens to use trigger mechanisms itself, it is impossible to add the triggers needed to establish the federation, instead they will have to be merged with the existing ones into a single complex trigger for each operation and table. In addition to that, triggers

[1]Persistence is a product of Persistence Software, Inc.

Table 1. Events and their Impact on Global Consisty

Event	Significance
DML operations (INSERT, UPDATE, DELETE)	Affect interdatabase dependencies (e.g. referential constraints, value dependencies)
Transaction commands (BEGIN, COMMIT, ROLLBACK)	Control the visibility of local updates to the global user
Stored procedures	Enable consistency rules fired by user-defined events
SELECT queries	Read (possibly) globally inconsistent data
DDL operations (table manipulation)	Violate the consistency between the local database and the global database schema

mean a substantial change of the local schema, which implies a loss of local autonomy.

Using a wrapping approach, the legacy code can be inspected with respect to possible consistency violations. An *application wrapper* surrounds complete legacy systems, both application code and data. Because of the individual nature of screens and functions, the development of application wrapping requires specialized and time-consuming solutions.

While application wrapping surrounds complete legacy systems (application code and data), *database wrapping* is based on establishing an interface between the application and the DBMS. This additional tier can process all incoming commands before their submission to the DBMS. Client-server DBMSs commonly allow to tap into the communication between clients and the database server. Therefore, they are predestined for a special database wrapping solution, viz. an application-independent gateway on top of the local DBMS.

Assessing the three mechanisms as sketched above, it can be stated that no solution is complete. However, in the current implementation, we proved the feasibility of our database wrapping approach by using the Sybase Open Server for the implementation of a gateway (see figure 2).

2.2 Architecture of the Gateway

The relational database gateway was implemented in Sybase using the OpenServer library. Figure 2 gives an overview of the components and their interactions. Besides the gateway components, there are some programs to initialize all databases that will be monitored by the gateway. The initialization comprises the creation of log tables and utility procedures, which has to be done bypassing the gateway. Furthermore, it serves for the configuration of the gateway depending on the kind of global consistency the user needs, for example, immediate or eventual consistency. The kind of service the gateway has to provide to the mediator can be specified. Figure 2 shows how the gateway together with the actual database server appears to the client applications, namely as complete SQL server where the gateway is transparent.

Incoming SQL statements or batches of statements are sent to the gateway, which directs them to an analyzer, instead of sending them directly to the database server. The analyzer works in two passes. In the first pass, a batch is split up into its statements. The analysis of single statements is done in the second pass. The result of the parsing step is processed in other components of the gateway. The logging component records each update event and enables inferring a database state history (see section 2.3). The signalling component is responsible for the communication with the mediator and enables the execution of a local command under control of the mediator (see section 2.4 for the description of the communication parameters). The logging functionality can be suppressed, e.g for performance reasons. The only purpose of the condensing component is to tie the log entries to net effects in the history. After the analysis of the original statement this command together with additionally created commands (in case of using the log component) is either redirected to the actual server in a straightforward manner or sent to the mediator (in case of global control of all statements). In [16] a complete description of the Sybase gateway is given.

2.3 Local Event Logging

The gateway provides a logging facility to enable the reconstruction of states that were valid at the detection time because it may happen that there is a time delay between the detection and the action to be triggered. These anomalies are also described in the literature about view maintenance [28] and are typical

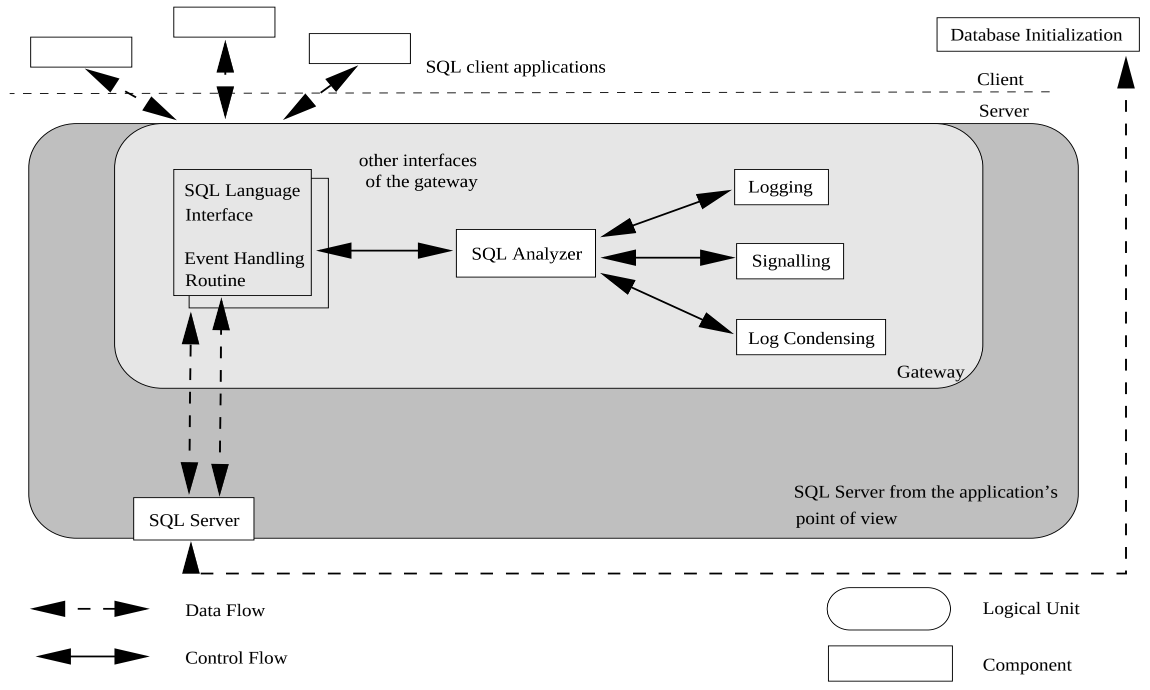

Figure 2. Gateway Architecture

in warehousing environments. We distinguish between extensional and intensional logging. Intensional logging only comprises the logging of the statements and the predicate that was taken from the WHERE clause. But intensional logging may not be sufficient because the interpretation of the predicate some time after the event occurence leads to different results if the database state has evolved meanwhile.

Hence, we provide extensional logging. The log tables are created from the original tables by extending them with additional columns (see table 2). By evaluating the WHERE clause before the actual execution of the statement, each affected tuple is stored in the log table.

The text of the original statements and the transaction commands are stored separately. To take into account the local transaction semantics, a transaction log table is maintained. Transaction statements are logged in the same manner as DML statements, the captured commands are begin, commit, rollback. The assignment of statements to transactions is based on their timestamps and the identifier of the client server connection. To reduce the number of entries in the transaction, a condensed log algorithm calculates the net effect of subsequent DML operations. The rules shown in table 3 illustrate how a new entry results from a combination of an existing one with a subsequent entry affecting the same tuple. They are similar to the differential snapshot refresh algorithm [14].

Table 2. Structure of Log Tables

Name	Semantics
column 1, 2, ..., n	user-defined semantics, taken from the original table *Table*
mod_time	time of the modification (system time)
mod_type	type of the modification: INS, DEL, UDE/UIN (inserted, deleted, old updated, new updated)
mod_thr	client identifier of the client server connection
mod_usr	login name of the user
mod_ts	timestamp (automatically incremented, serves to order operations)

Deviating from this algorithm, a delete entry cannot be aggregated with a subsequent insert: Due to an object-oriented view this cannot be interpreted as an object modification because an object with a new object identity is created after an insert.

Table 3. Condensation Rules

Log Entry	Condensed Entry			
	no entry	delete	insert	update
delete	delete	-	no entry	delete
insert	insert	-	-	-
update	update	-	insert	update

2.4 Local Event Signalling

The gateway can be seen as part of the heterogeneous system, responsible for the communication with the mediator which is capable of reacting to local events. The signalling is performed similar to the generation of the log entries as described in section 2.3. It is activated by the SQL analyzer and works by remote procedure calls with a parameter data structure describing the information about statements to be monitored. The parameters depend on the statement type. The current implementation only considers the SQL DML statements INSERT, DELETE, UPDATE. Figure 3 shows the current parameter data structure.

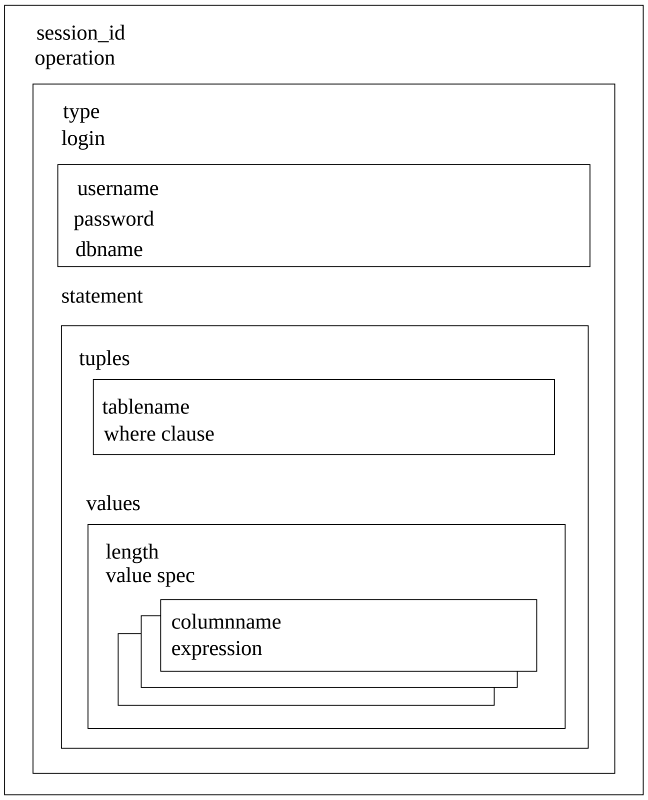

Figure 3. RPC Parameter Data Structure

3 Bridging the Gap Between Relations and Objects

3.1 Structural Forward Mapping from Objects to Tables

In most multidatabase projects during the last years, the object-oriented data model has been chosen as global data model: Class structures, inheritance capabilities and the concept of polymorphism allow to capture significantly more semantics than the relational model. Objects encapsulate the implementation of the data and methods and separate implementations from interfaces, and therefore an object-oriented model provides a natural mechanism for the translation to and from other data models.

Up to now, there has been little agreement on an object-oriented data access and query language, but there is some standardization effort in progress by the ODMG [5]. According to the ODMG specification, there is a mapping from the ODMG model constructs to the C++ programming language. Hence, it appears obvious to choose a C++ based model, as this also is the global data model provided by the vast majority of OODBMS vendors.

At the same time, relational databases remain the dominant data repositories, and as such have to be integrated into federated databases. Looking for solutions that link C++ based systems to existing relational databases, two different approaches have been taken: Some OODBMS vendors offer additional tools to access relational data from C++ applications (such as ObjectStore's SQL Gateway), while alternatively, there are a number of object- relational systems available (e.g. UniSQL [24], Subtleware [23] and Persistence [20]), enabling the user to build object-based applications using relational systems as a storage base for their persistent data.

3.2 The Persistence Product: Overview and Special Features

The Persistence product provides development tools that enable object-oriented applications to access industry standard relational database management systems (RDBMSs), such as Sybase, Oracle, Informix or Ingres through a C++ interface. Persistence consists of two major functional components: the Relational Interface Generator (RIG) which generates portable, database independent C++ classes mapped to relational data and the Relational Object Manager (ROM), implementing database access and maintenance (object integrity and transaction management) [15].

Given the appropriate object model, the Persistence RIG creates a C++ class for each object in the model. Every class implements its own set of methods for database interface, such as create persistent or transient object instances, set or update object attributes or relationships, query using object-oriented features or ANSI SQL and delete objects from memory or database. Inheritance of attributes, methods and relationships, particularly propagating superclass queries to subclasses, and the usage of virtual methods to support polymorphism are also supported.

Persistence maps associations to foreign keys in the relational database, offering methods to access associated objects through the defined relationship. Thus, the application programmer can transparently navigate class associations or aggregations without knowing how they are implemented in the database.

3.2.1 Accessing Multiple Databases Simultaneously

The key feature that makes Persistence suitable for the implementation of multidatabase mediators is its flexibility as far as accessing multiple databases is concerned. Through the design of its class hierarchy, each Persistence-generated database class may be mapped to a particular database system connection (*PS_Connection*) and table. The mapping may be even changed dynamically while an application is running. For example, an application could open connections to an Oracle and a Sybase database, mapping the *Customer* class to an Oracle table while mapping the corresponding *Account* class to Sybase [15].

3.2.2 Reverse Engineering of Relational Databases: The Dictionary Reader

Persistence can read object model information directly from the RDBMS's data dictionary or system tables, allowing to automatically create C++ classes that correspond to existing tables. The data dictionary reader filters information in the RDBMS's system tables into a Persistence project structure: For all user tables in the database, a corresponding Persistence class with the table's columns as attributes of the appropriate data types is created. Primary key information is used to determine Persistence key objects (these are special objects containing the primary key information maintained by Persistence), and foreign key information to set up relations between participating classes. In the current release, there are still some restrictions, as far as determining correct cardinalities for the resulting relations is concerned: Regardless of e.g. unique or not null definitions and intermediate tables, every relationship is set up to be a zero-or-one (source class) to zero-or-many (destination class) relationship.The reverse engineering function of Persistence is restricted to structures and, hence, does not comprise a mapping of SQL operations.

4 Behavioral Mapping of SQL Statements to C++ Method Calls

The reverse mapping of SQL statements to C++ method calls (not provided by Persistence) has to face two serious problems resulting from the mismatch between a dynamic interpretative query language like SQL and the static compile-dependent approach of a C++ method based interface: The first problem is caused by the fact that the SQL language integrates the two very different tasks of data definition and data manipulation. From the present point of view, mapping the data definition language (DDL) command subset of SQL to C++ methods seems impossible. Even the most simple schema modification DDL command (such as a create table) would have to be mapped to the creation of a new class in the C++ based data model, a fact, which requires all the applications based on the corresponding schema to be recompiled, thus making an automatic mapping of SQL DDL commands completely impossible. A possible work-around for this problem we are exploring is the use of notification rules to signal local schema modifications.

Although the mapping of data manipulation language (DML) commands to C++ methods is possible, as long as the underlying database schema is static, the second problem arises when trying to create a database independent SQL to C++ mapper process: The source code of such a translator program will have to work on the classes and attributes generated from the database schema of a particular database, and as such be absolutely dependent on the current database schema. This problem will be addressed in section 6, where the concept of the mediator generator will be described.

Aside from these problems, the mapping of SQL DML statements accessing a particular database table to the equivalent method calls on the corresponding class in the equivalent C++ data model conforming to the Persistence generated method interface is just straightforward:

The SQL to C++ translator prototype we have developed currently supports (in addition to a CONNECT and DISCONNECT operation managing database login and logout) only a subset of all possible SQL DML commands, that is, the transaction management commands BEGIN, COMMIT and ROLL-

Table 4. Mapping of SQL Statements to Equivalent Persistence Methods

SQL Statement	Persistence Method Call
CONNECT	Constructor PS_Connection()
DISCONNECT	Destructor PS_Connection()
BEGIN	beginTransaction()
COMMIT	commitTransaction()
ROLLBACK	rollbackTransaction()
INSERT	Constructor <Class>()
UPDATE	set<Attribute>()
DELETE	remove()
SELECT	selectMany()

BACK as well as the database manipulation commands INSERT, UPDATE and DELETE and the SELECT database query command[2]. An overview of the currently supported DML commands is presented in table 4.

While the implementation of the CONNECT and DISCONNECT operations as well as the three transaction management operations is not very demanding, the remaining operations are somewhat more tricky. The complete description can be found in [18].

Having retrieved and fully parsed a SQL DML command on a particular class, our translation algorithms work as follows: For each class in the application schema Persistence generates a key object class which contains each attribute of the primary key of that class as data member. When creating a new object instance of the application class, a corresponding key object is created at the same time to ensure Persistence object cache consistency. The special treatment of primary and foreign key attributes is needed because Persistence does not allow to change key attribute values directly through a $set\langle Attribute\rangle()$ method.

INSERT

1. Create a new non-persistent object of the respective class.

2. Retrieve the corresponding Persistence key object.

3. In a loop for all attributes referenced in the INSERT statement:

    4. **if the attribute is a primary key attribute**, call the appropriate $set \langle Attribute\rangle()$ method on the key object;

       if the attribute is a foreign key attribute, retrieve the related object out of the corresponding class using the foreign key value, then call the $set\langle Relationship\rangle()$ method on the new object just retrieved as a parameter;

       if the attribute is a non-key attribute, call the appropriate $set \langle Attribute\rangle()$ method on the newly created object.

5. Store the updated key object into the new object.

6. Make the new object persistent using the $\langle Class\rangle :: insert()$ method.

[2] Statements to be supported in the future enclose stored procedure calls and administration commands.

UPDATE

1. Retrieve all objects from the particular class, for which the SQL WHERE expression holds into a collection of objects ($\langle Class\rangle\_Cltn$) from this class, using the Persistence $\langle Class\rangle :: query SQLWhere()$ method.

2. In a loop, for all objects in this collection:

    3. In a loop over all attributes contained in the SQL SET value assignment part:

        4. **if the attribute is a primary key attribute**, create a non-persistent object (NPO) as a deep copy of the current (persistent) object (PO), then change the primary key attribute value in the NPO to the desired value using the $set\langle Attribute\rangle()$ method on the NPO, make the new NPO persistent and remove the old PO from the database;

           if the attribute is a foreign key attribute, retrieve the corresponding related object with the new value for the key attribute from the related class using the $\langle RelatedClass\rangle :: queryKey()$ method, then call the $set\langle Relationship\rangle()$ method on the current object using the object just retrieved as parameter for the call;

           if the attribute is a non-key attribute, just call the appropriate $set\langle Attribute\rangle()$ method.

DELETE

1. Retrieve all objects from the particular class, for which the SQL WHERE expression holds into a collection of objects ($\langle Class\rangle\_Cltn$) from this

class, using the $\langle Class \rangle :: querySQLWhere()$ method.

2. In a loop for all objects in this collection:

   Remove the object using the $\langle Class \rangle :: remove()$ method.

5 The Mediator Architecture

To deal with data from multiple sources, it is necessary to apply an intelligent processing mechanism capable of extracting globally accessible data using aggregation or selection operators. On the other side, when querying data in a multidatabase environment, semantic conflicts regarding different names, abstraction levels and structures have to be solved, called the context interchange problem in [19]. Global consistency constraints are also a kind of knowledge that goes beyond the local applications. The gap between the globally available data and the useful information may grow in the future and needs solutions incorporating knowledge about the local data. This knowledge can be implemented in an additional software layer acting as a mediator between the end user applications and the databases. In [26], some tasks a mediator can execute are presented. Among them we focus on the problem of constraint management implemented using ECA rules.

Considering our sample environment, we have relational DBMSs and some clients. The Persistence-based mediator process behaves just as a usual Sybase (or Informix) client. Having wrapped the database server by the gateway, whenever a query is submitted by a local client, it will be signalled to the mediator process, translated and executed against the local DBMS. By executing locally submitted SQL commands as C++ method calls, the consistency between the object cache and the database is automatically maintained by the Persistence ROM. The main problem to be solved is the passing of the results in case of a SELECT statement from the mediator process back to the gateway process, so that the retrieved tuples are made available to the local client. Figure 4 illustrates the data flow at the submission of a local query.

6 The Mediator Generator

As stated in chapter 4, the main conceptual problem when trying to make the implementation of a mediator process independent from a particular database schema (or Persistence object model, resp.) is the transition from the (possibly dynamic) SQL statements to the static method interface generated by Persistence. Facing this fact, we realized that, in order to be able to implement a SQL to C++ mapping, we had to confine ourselves to an underlying, static database schema in the form of a Persistence object model, for which we could create an object model specific mediator application.

To store the assignment of each class of the object model to a DBMS participating in the federation a connection map file was introduced. We then developed the idea of automatically generating such an object model specific mediator by a specially designed "mediator generator" application, which should take the Persistence object model and connection mapping information as input and generates the source code of a Persistence based C++ mediator application for this particular object model. SQL statements issued by local database applications or interactive users should be intercepted by the SQL gateway described in chapter 2, and then, in spite of being directly processed by the SQL server, passed to the generated mediator through the remote procedure call (RPC) interface shown in section 2.3 (RPC XDR protocol interface file *exec_dml.x*). The resulting mediator is a RPC server, typically running as a background process, which waits for RPC requests issued by its clients (the SQL gateways), distinguishes them by the type of SQL command to be executed and then does the appropriate translation from SQL to C++ according to section 4.

The mediator generator consists of two main modules: a lex/yacc-generated parser for Persistence object models, which filters the needed information about the classes, attributes and relationships from the Persistence object model description file, and a preprocessor, which simply replaces a set of about 30 different preprocessor instructions in a series of mediator source code templates, particularly the main mediator RPC server module template *Server.C.tmpl*, with the object model information just extracted.

There are three types of preprocessor instructions:

- pure *replacement* instructions simply replacing the instruction with information gained from the object model;
- *loop* instructions directing the control flow of the preprocessor through the input template;
- *condition* instructions, which copy or skip source code blocks from the input template depending on whether some given conditions hold or not.

Using the loop instructions, for both classes and attributes, a special source code block gets copied to the output file once for each class or attribute of a class, respectively. In parallel, class and attribute indices

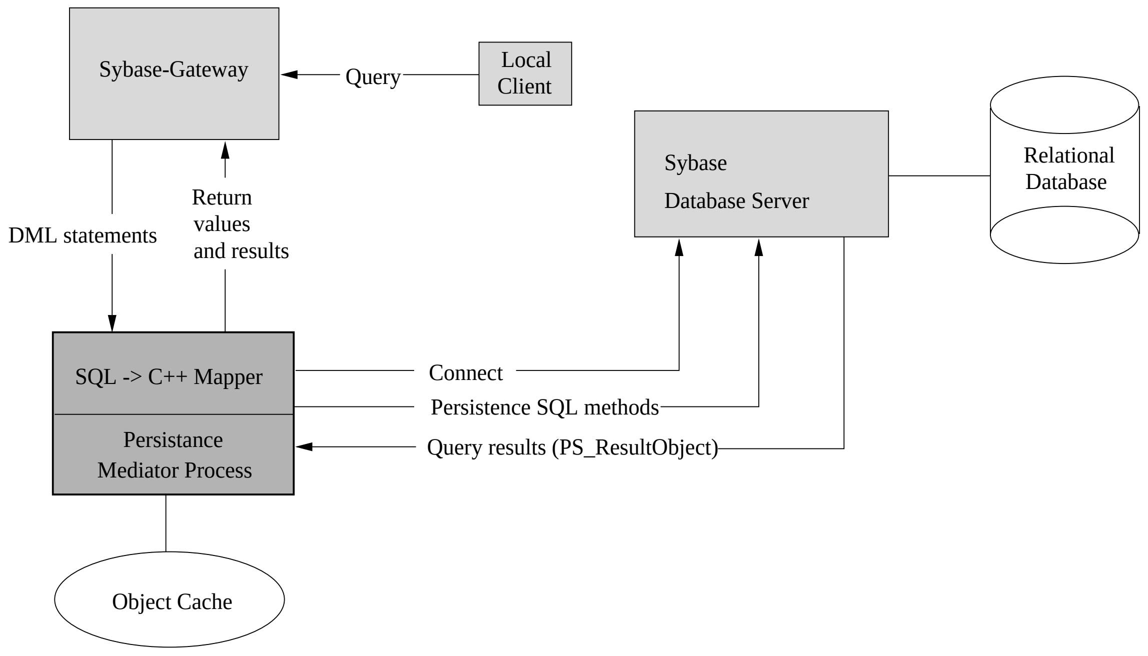

Figure 4. Communication Between Gateway, Mediator Process and Local Database Server

are increased, such that the replacement instructions within a loop code block get replaced at each pass by the name of the next class or attribute, respectively. The condition instructions allow to copy or skip blocks from the source code, for example in order to differentiate between the handling of primary, foreign and non-key attributes when updating an attribute (cf. section 4).

The complete structure of the mediator generation process with all participating programs, files and libraries is shown in figure 5.

7 Active Capabilities of the Mediator

7.1 Active Processing: Hooks

Persistence offers the possibility to define notification hooks on classes, attributes and relationships. A notification hook is a C++ method defined on the respective class which is called automatically whenever the relating event will take place next (*pre hook*) or has just taken place (*post hook*). Hooks are available on the creation, removal, query and modification of objects in a class, the change of an attribute's value or the change of a relationship or query of related objects.

When the Persistence RIG generates the code for an object model, empty stubs for each of the defined hooks are generated that can easily be extended to execute every piece of code the user implements for the method. In particular, it is possible to check whether a set of conditions holds before taking a specific action. The Persistence hooks can therefore be used to implement the functionality of the standard ECA (event, condition, action) rules [11] or the triggers in relational systems, respectively. Although the coupling modes for simple C++ hooks are immediate (for example, the actions in an if-block are executed immediately after the condition has been checked), it is nevertheless possible to realize even the more complex deferred or detached coupling modes [3] using threads and interprocess communication features.

7.2 Event Processing in a Heterogeneous System

In a first attempt, rules were implemented using hook methods. The next step we are working on is the implementation of a rule management component defining rules as subclasses of a common superclass *Rule*. Following this approach, the hooks only detect the event and activate the rule management module by calling a raise method describing the event type together with its parameters. The rules are part of the metadata maintained by the federated system and accessed by the mediator.

As stated in section 1, in a federated system, we have to distinguish between local and global events. We want to define three parameters serving the classification of different rule execution scenarios in an active

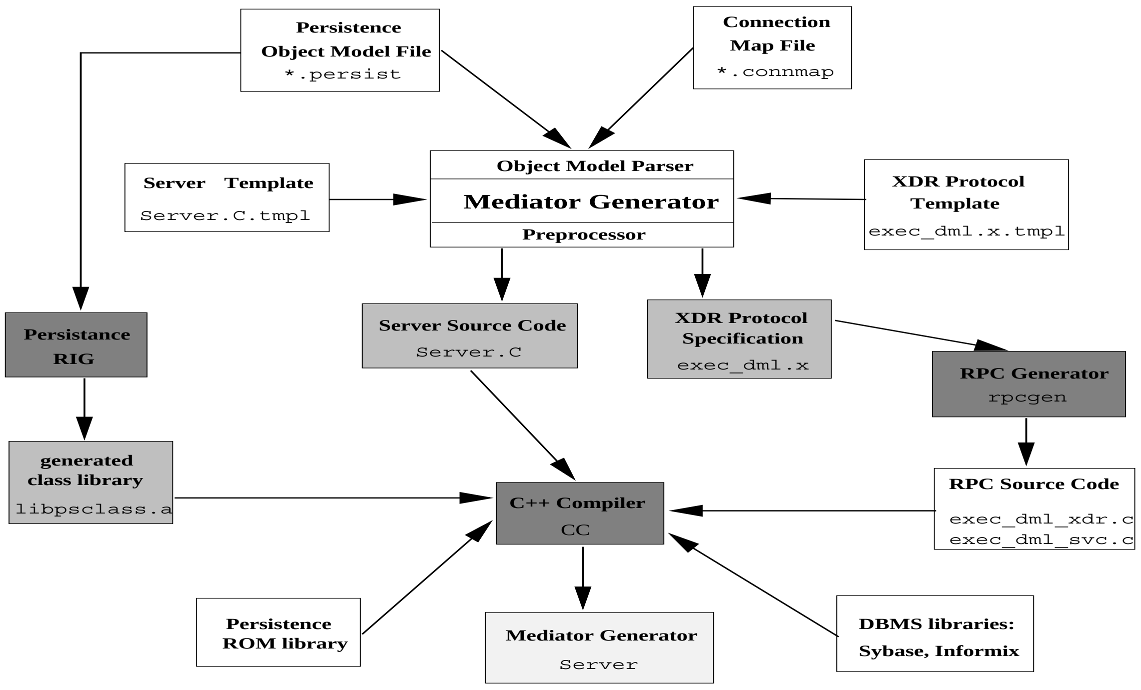

Figure 5. Overview of the Mediator Generation Process (Programs, Files and Libraries

federated system. To do this, we take database events as an example.

(a) Events may have a local or a global origin or mode of submission, which can be through the local API or the global C++ interface, dependent on the user who caused the event.

(b) The scope of database operations may be limited to a single database or to multiple databases. With respect to our object-relational system, a DML operation may be executed as a SQL command only at the local API (local scope) or as method call in the Persistence-based system (global scope), regardless of the origin of the event. In the presence of a database gateway, locally submitted statements can be directed to the global system to avoid the existence of real local operations unknown to the global system that controls everything.

(c) The events may be recorded in a server event log independently from the kind of application or user. That presupposes the existence of a gateway component on top of the database server recording each incoming statement in the event history. In case of statements with global scope the gateway functionality is restricted to logging as sketched in section 2.3.

Example: local event origin, global scope, with server event log
A table *Person(name,nickname,age)* is given. A local user wants to delete all tuples matching the predicate *"age > 30"*. Using the local SQL interface, the user issues the command *DELETE FROM Person WHERE age > 30*, i.e. the origin of the command is local. Because we assumed a global scope, the gateway does not forward the statement immediately to the local database server, but to the mediator process, where it is translated to a sequence of Persistence C++ method calls, and executed against the local database server or the surrounding gateway. That means, the class method of the equivalent class $Person :: querySQLWhere("age > 30")$ returns a set of Person instances matching the WHERE clause. Subsequently, on each retrieved object in the set, the remove method is called which actually deletes the object from the relational database. If it was defined in the gateway configuration, server log tables are written, namely the statement log table and the table *Person_Log* with all deleted tuples (cf. section 2.3). Table 5 gives an overview of combinations that are conceivable in our object-relational system.

Table 5. Event Handling in a Heterogeneous System

a) origin/mode of submission	b) scope	c) server event log
local	local	no
local	local	yes
local	global	no
local	global	yes
global	global	no
global	global	yes

8 Conclusions

The presented solution is a promising approach to combine features of multidatabase systems and active database systems when integrating multiple relational databases. The mediator processes that can be generated due to our algorithms are each specific to a certain multidatabase schema. Enhancing the mediator with rule management components allows to express more complex consistency requirements.

On the other side, the local systems's willingness to make local operations available to the federation is one of the main prerequisites in order to guarantee global constraints with a high quality. The gateway we have presented is tailorable with respect to consistency or performance requirements. First performance measurements showed encouraging results, a significant performance loss only occurs if the local log facility is activated, otherwise the response time delay when incorporating the mediation system is negligible. Therefore, the configuration can be determined dependent on the desired functionality and performance needs. For example, the monitoring or logging is restrictable to certain tables of global relevance. Thus, deviations from a globally consistent state may be tolerated if the events can be reconstructed later using the gateway's log. Hence, we believe that, due to its flexibility in the configuration of the gateway and the mediator, our approach can be used as a practical platform to explore the tradeoff of local autonomy and the enforcement of global constraints.

Acknowledgements

The authors are grateful to Persistence Software, Inc. for providing a license of the Persistence system and to Reinhold Krayer who implemented the database gateway as a part of his Masters Thesis.

References

[1] R.M. Alzahrani, M.A. Qutaishat, N.J. Fiddian, W.A. Gray; *Integrity Merging in an Object-Oriented Federated Database Environment,* in: 15th British National Conf. on Databases, 1995.

[2] A. Buchmann; *Modelling Heterogeneous Systems as a Space of Active Objects,* in: Proc. 4th Intl. Workshop on Persistent Objects, Martha's Vineyard, Sept. 1990.

[3] H. Branding, A. Buchmann, T. Kudrass, J. Zimmermann; *Rules in an Open System: The REACH Rule System,* in: Proceedings of the First International Workshop on Rules in Database Systems, Edinburgh, 1993.

[4] A. Buchmann, J. Zimmermann, J.A. Blakeley, D.L. Wells; *Building an Integrated Active OODBMS: Requirements, Architecture and Design Decisions,* in: Proc. 11th Internat. Conference on Data Engineering, Taipeh, 1995

[5] R. Cattell (ed.); *The Object Database Standard: ODMG-93,* Morgan Kaufmann, 1993.

[6] S. Ceri, J. Widom; *Managing Semantic Heterogeneity with Production Rules and Persistent Queues,* in: Proceedings of the 19th International VLDB Conference, 1993.

[7] S. Ceri, P. Fraternali, S. Paraboschi, L. Branca; *Active Rule Management in Chimera,* in: J. Widom, S. Ceri (eds.): Active Database Systems: Triggers and Rules For Advanced Database Processing, Morgan Kaufmann, 1996.

[8] M. Castellanos, T. Kudrass, F. Saltor, M. Garcia-Solaco; *Interdatabase Existence Dependencies: A Metaclass Approach,* in: Proc. of the Internat. Conference on Parallel and Distributed Databases (PDIS), Austin, 1994.

[9] S. Chakravarthy, V. Krishnaprasad, Z. Tamizuddin, R.H. Baddani; *ECA Rule Integration into an OODBMS: Architecture and Implementation,* in: Proceedings of the 11th Internat. Conference on Data Engineering, Taipeh, 1995.

[10] C. Collet, T. Coupaye, T. Svensen; *NAOS - Efficient and Modular Reactive Capabilities in an Object-Oriented Database System,* in: Proceedings of the 20th International Conference on Very Large Databases (VLDB), Santiago, Chile, 1994.

[11] U. Dayal; *Active Database Management Systems,* in: Proc. of the 3rd Internat. Conference on Data and Knowledge Bases, Jerusalem, 1988.

[12] S. Gatziu, K.R. Dittrich; *Events in an Active Object-Oriented Database System,* in: Proceedings of the 1st Internat. Workshop on Rules in Database Systems (RIDS'93), Edinburgh, 1994.

[13] N.H. Gehani, H.V. Jagadish; *Ode as an Active Database: Constraints and Triggers,* in: Proc. of the 17th Internat. Conference on Very Large Databases (VLDB), Barcelona, 1991.

[14] B. Kähler, O. Risnes; *Extended Logging for Database Snapshot Refresh,* in: Proc. of the 13th International Conference on Very Large Data Bases (VLDB), Brighton, 1987.

[15] A. Keller, R. Jensen, S. Agarwal; *Enabling the Integration of Object Applications with Relational Databases,* Persistence Technical Overview, Persistence Software, Inc., 1994.

[16] R. Krayer; *Entwicklung eines relationalen Datenbank-Gateways zur Unterstützung globaler Konsistenzkontrolle (Development of a Relational Database Gateway to Support Global Consistency Control),* Masters Thesis (in German), Tech. Univ. Darmstadt, Dept. of CS, 1995.

[17] Q. Li, D. McLeod; *Managing Interdependencies among Objects in Federated Databases,* in: Proc. of the DS-5 Semantics on Interoperable Database Systems, Lorne, Australia, 1992.

[18] A. Loew; *Evaluierung des Datenbank-Integrationstools Persistence and Erprobung als aktives Vermittlersystem (Evaluation of the Database Integration Tool Persistence and Usage in an Active Mediation System),* Masters Thesis (in German), Tech. Univ. Darmstadt, Dept. of CS, 1995.

[19] S.E. Madnick; *From VLDB to VMLDB (Very Many Large Databases): Dealing With Large Scale Semantic Heterogeneity,* Proc. of the 21st International Conference on Very Large Data Bases (VLDB), Zurich, 1995.

[20] Persistence Software, Inc.; *Persistence User Manual for Release 2.3,* 10/94.

[21] A. Sheth, J.A. Larson; *Federated Database Systems for Managing Distributed Heterogeneous and Autonomous Databases,* ACM Computing Surveys 22(1990), 3.

[22] A. Sheth, M. Rusinkiewicz; *Management of Interdependent Data: Specifying Dependency and Consistency Requirements,* Proc. Workshop on Management of Replicated Data, Houston, 1990.

[23] *Subtleware Database Technology Connectivity,* WWW page: http://world.std.com/ subtle/info.html, Subtle Software, Inc., 1995.

[24] *UniSQL's Object-Relational Data Management Technology,* Enterprise Reengineering Product Profile, The Bowen Group, Ferndale, WA, 1995.

[25] K. Vanapipat, N. Pissinou, V. Raghavan; *A Dynamic Framework to Actively Support Interoperability in Multidatabase Systems,* in: Proc. of the 5th Internat. RIDE-Workshop on Distributed Object Management, 1995.

[26] G. Wiederhold; *Mediators in the Architecture of Future Information Systems;* IEEE Computer March '92.

[27] P. Winsberg; *Legacy Code: Don't Bag It, Wrap It,* in: Datamation, May, 1995.

[28] Y. Zhuge, H. Garcia-Molina, J. Hammer, J. Widom; *View Maintenance in a Warehousing Environment,* in: Proceedings SIGMOD Internat. Conference on Management of Data, San Jose, 1995.