THE ANNALS OF "DUNAREA DE JOS" UNIVERSITY OF GALATI
FASCICLE III, 2004 ISSN 1221-454X
ELECTROTEHNICS, ELECTRONICS, AUTOMATIC CONTROL, INFORMATICS

VENDOR-INDEPENDENT DATABASE APPLICATIONS – AN ARCHITECTURAL APPROACH

Octavian Paul Rotaru ** Marian Dobre ** Mircea Petrescu *
** Computer Science and Engineering Department, University "Politehnica" of Bucharest, Romania*
*** On leave of absence from *, Currently at Amdocs Dev. Ltd., Limassol, Cyprus*
Octavian.Rotaru@ACM.org, Marian.Dobre@amdocs.com, MirceaStelian@yahoo.com

Abstract: The ability to switch between different Database Management Systems (DBMS) is a requirement for many database applications in which effort was invested by many researchers. The main obstacle is the non-uniformity across vendors of the SQL language, the de-facto standard in the industry. Also, an application that maps between an object-oriented application and a relation database needs to be designed in a proper way, in order to achieve the required level of performance and maintainability. This paper presents, extends and further details the Vendor-Independent Database Application (VIDA) framework, initially proposed by us in [9]. The proposed VIDA architecture is described in-depth, based on our practice and experience in this field. The design decisions are presented along with supporting arguments. The VIDA architecture presented here aims to fully decouple the application both from the query language and from the database access technology, providing a uniform view of the database. The problems encountered, both during design and implementation, are presented along with their solutions. Also, the available data access technologies and languages are surveyed and their conformity with a standard is debated.

Keywords: *database access layer, DBMS, design patterns, SQL standard, vendor-independence*.

1. INTRODUCTION

Ideally we would like applications to be able to use each and every of the data sources required, irrespective of their vendor, version and particular constraints. Practically, it is very difficult to implement a fully vendor-independent database application. Usually, most of the database applications intended to be vendor-independent are finally strongly linked to the database engine that was used during development. This happens due to performance issues that cannot be solved without using some vendor specific features, or because of the insufficient analysis, design and development time allocated.

In order to achieve vendor-independency, a database independent *data provider* module is required to assure a uniform way for connecting to, retrieving from and saving information into the data source. Implementing a database-independent data provider brings into the picture the issue of choosing the proper database access protocol for the application being implemented. Even if such a data provider is available on the shelf or can be implemented with a reasonable effort, this does not solve all the associated problems, as described below.

Structured Query Language (SQL) is the dominant language for data retrieval and manipulation in databases. An important issue related to its use in VIDA is that there is no fully SQL standard compliant database. Different vendors use a different syntax for the same concept. The syntax of SELECT, INSERT and UPDATE (part of DML – Data Manipulation Language) is to a great extent similar, but the Data Definition Language (DDL) statements and joins have different syntax across platforms.

For most of the applications, sticking to the common kernel of the SQL DML syntax and using an independent data provider is a solution ensuring that the application is database vendor independent. However, in case of applications having special requirements like for example to create temporary or permanent tables, modify their structure or drop them, join huge amounts of data that cannot be done into the memory, the solution has to be extended in order to also cover these aspects.

Fully decoupling the application from the database by using a *mediation layer* along with the *data provider* is in most of the cases the best solution, but sometimes this proves to be not very easy to implement. Using a *class factory* in order to obtain a

This paper was recommended for publication by Severin BUMBARU

proper SQL object for the database to which the application is connected is the design solution used in our framework for VIDA. The above-mentioned SQL object is actually a *statement composition tool* customized for each and every database engine that our framework supports. However, adding a new DBMS to the list of the supported ones triggers the recompilation of the mediation layer. Finding a component based plug-and-play solution for implementing the mediation layer is a future research objective.

2. THE ANSI SQL STANDARD

Having a standard is beneficial for everyone. The conformity with the standard gives to the users the assurance that the product does what is supposed to do. A utility or a tool designed compliant with the standard will work on all the available DBMS, irrespective of their vendor. Similarly, programmers can become certified in the standard and not in a vendor-specific implementation [10].

In 1989, American National Standard Institute (ANSI) published the first SQL standard specification, intended to make it independent of a specific implementation or DBMS. Since then, the SQL standard was twice revised in 1992 (SQL-2 or SQL-92) and in 1999 (SQL-3 or SQL-99).

Up to this point everything looks right. There is a standard, so why to worry? Just implement your application according to the standard and it will be vendor-independent. Looks obvious, isn't it? Unfortunately it is not, mainly because there is no DBMS fully compliant with the SQL standard. Various database manufacturers have taken allowances with the ANSI/SQL standard to different degrees in order to give their product a competitive advantage and meet customer demands [10]. These vendors interpret the standard in their own way, using proprietary syntaxes in some cases and adding new extra features. Every DBMS vendor wants to differentiate its DBMS product and to have its own SQL "savor". Apart from supporting most of the ANSI/SQL standard, there are always features, enhancements or extensions that are available only from individual vendors. Practically, the number of SQL dialects equals the number of DBMS vendors.

Using these additions to the standard in an application is on the long run a very bad decision. The portability is gone, unless the application code is changed in order to come back to the standard, and the quality suffers as well. Even though these extra features seam to be useful and to shorten the development time, their use is strongly linking the application to the database vendor.

Considering the above-mentioned facts, a question arises: is SQL a standard in this moment? Most probably not, or not in the way it was intended. Actually, SQL evolved from a standard to general guidelines, most of the database manufacturers considering the compliance with it as secondary. The main goal of the vendors is to get closer to the SQL standard compliance without sacrificing speed or reliability. The non-standard features are considered as greatly increasing the usability of their products. Some famous examples in this sense are the Oracle plus (+) syntax for outer joins and the TOP syntax available in the Microsoft products. Even if they are shorter and easier to use than the standard notation, they still remain a deviation from the SQL standard.

If an application using only DML statements can be vendor-independent by simply avoiding the usage of non-standard features, this does not hold in the case of an application issuing also DDL statements. If the syntax for CREATE and DROP TABLE is usually the same, excepting the data types, the syntax of ALTER TABLE is usually different from one vendor to another.

In the recent years, more and more database manufacturers make steps towards compliance with the SQL-92 and SQL-99 standard. Even so, there are still many things to be done until it will be possible to easily change the database used by any application, no matter the way it was implemented.

3. CHOOSING A DATABASE ACCESS PROTOCOL

General *database access protocols* were designed so that the detailed information about a particular database engine can be "snapped in" a common framework without worrying about the implementation's specifics [6].

Open Database Connectivity (ODBC) is definitely the most used database access protocol. Even if ODBC is slower than some newer technologies like OLE-DB and ADO, it has the widest support of both databases and applications. OLE-DB is probably the highest performance protocol for accessing a database, but it is limited to Windows platforms. JDBC, released by Sun shortly after Java, is limited to Java applications. Hence, ODBC is the only platform and language independent generic database access protocol available.

Due to the above-mentioned reasons, our choice was to use ODBC as the generic database access protocol for our VIDA framework. However, in order to decouple the application from the data provider, the protocol-dependent connectivity details should be accessed through an *abstract interface*. In this way, the database access protocol can be changed at any moment by simply adding a new implementation for it. Only one layer (component of the application) is affected.

Creating an *abstract protocol-independent connectivity layer* is useful in terms of reusability. It can be used by many applications irrespective of the database access protocol, and therefore we identified it as an important component of our vendor-independent database application architecture.

More details about the framework architecture are provided in Section 7 of this paper.

4. HANDLING DDL STATEMENTS

DDL is the biggest issue to be addressed when speaking about applications able to switch from a DBMS to another. This is mainly because the high diversity of DDL SQL dialects offered by various vendors.

ODBC defines its own data types, which are used both for data definition and data manipulation. The database also has its own native types and therefore a mapping is needed.

The solution provided by ODBC to this issue is the *SQLGetTypeInfo* function used to retrieve the data type information. This is not solving the problem completely, because the relation between ODBC types and database types is not bijective.

In case there is no corresponding native data type for an ODBC one, a solution is to check for all the compatible ODBC types if they have a native correspondent. The search ends if a mapping is found or there is no unchecked possible compatibility left.

Finding the mapping needs to be done through a module that will try all the similar types based on a compatibility diagram. A state transition diagram for ODBC data types compatibility is shown in Figure 1.

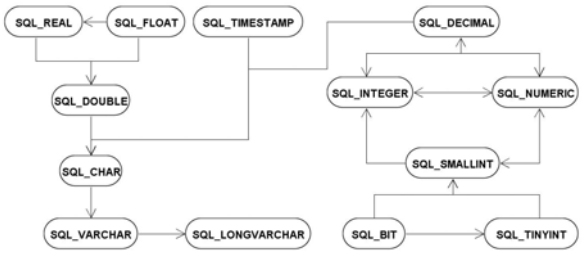

Fig.1. ODBC Data Types Compatibility State Transition Diagram

For example, in case a certain database does not have a correspondent for SQL_NUMERIC, the system will try to see if any of the compatible types has a native correspondent. In this case, the compatible types are SQL_DECIMAL, SQL_INTEGER and SQL_SMALLINT. In case none of these numerical equivalences is successful, a SQL_CHAR conversion will also be tested.

As shown in Figure 1, visiting all the possible states (nodes), starting from the state corresponding to the ODBC type, for which compatibilities are searched, enables to find all its compatibilities. For every compatible ODBC type *SQLGetTypeInfo* is used in order to check if a corresponding native database type exists.

Since the state transition diagram in Figure 1 can be visited starting from any of its states, there is no explicit start state defined.

The ODBC data types compatibility was initially implemented as a matrix of Booleans. The ODBC data types were placed on both axes, and a true value at the intersection of two types used to indicate a possible conversion.

In order to always have the best conversion possible, an extra parameter was used for ODBC type conversions: priority. Therefore, the initial solution using a conversion matrix was discarded and the ODBC mapping was implemented using a priority list, as depicted in Figure 2.

Finding the best existent compatible mapping for an ODBC type is done by visiting the nodes of its associated linked list until an ODBC type having a native database type mapping is found or the end of the list is reached.

Reaching the end of list will trigger a type incompatibility exception. However, our ODBC data types compatibility tests shows that such a situation is very improbable to appear. We never encountered such a situation while testing on different version of MS Access, MS SQL Server, Oracle, Sybase and Informix.

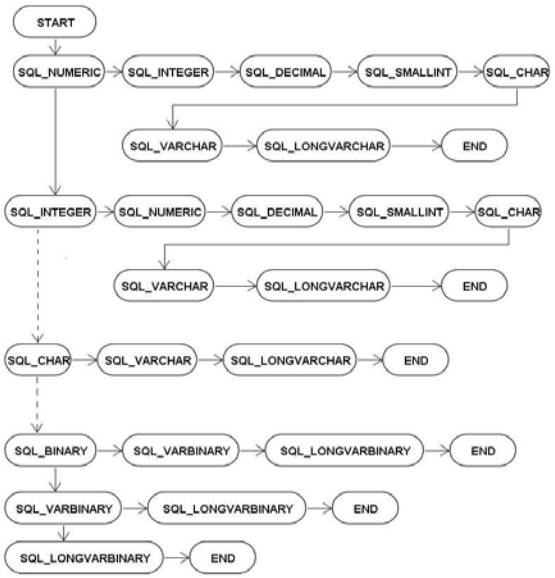

Fig.2. ODBC Data Types Conversion Priority Diagram

5. THE STATEMENT COMPOSER – WHY AND HOW

As discussed in section 2, it is not possible to rely on the SQL standard in order to achieve database platform independence. This affects even the DML statements that are usually the most generic of all.

The solution proposed by our framework is a SQL *statement composition module* able to create the database specific statement based on meta-data provided by the client application or the signature of the database obtained through the database access protocol. In this way the client application is potentially isolated not only from the SQL dialect, but also from the query language. Since the statements are composed by the framework and not by the client application, the client is not at all aware of the query language. The framework was implemented for SQL only, but the query language isolation also provided by the query composer gives us the possibility to add the required support for any other query language without affecting the application, but only the underlying database access framework.

Probably the best examples of vendor specific statements are ALTER and JOIN syntaxes. The ALTER statement is the most vendor-dependent of all. It looks like all the database vendors tried their creativity on it. These kinds of examples justify our decision to develop a *statement composition tool* able to "translate" the conceptual statement into a real one, in concordance with the SQL dialect of the database, in order to handle syntax diversity. .

The implementation of the statement composer was done according to the conceptual diagram presented in Figure 3.

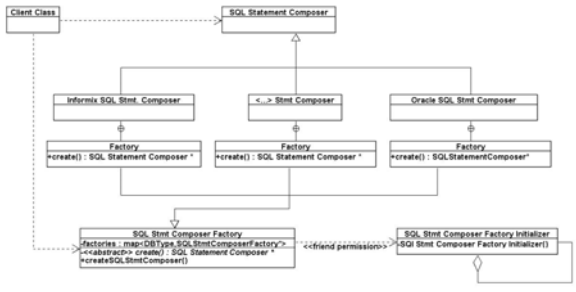

Fig.3. The Class Diagram of the Statement Composer (implemented using the class factory pattern)

The implementation of the statement composer was based on the *class factory design pattern*. <SQL Stmt Composer> is the generalization of the <Oracle SQL Stmt Composer>, <Informix SQL Stmt Composer> and/or any other vendor specific statement composition class. <SQL Stmt Composer Factory> class is a generalization of the inner Factory classes contained in all the statement composition classes. The inner Factory classes implement the *create* operation, declared as abstract in the generalization, used to create a new instance of the outer class.

6. JOINS – PERFORMANCE CONSIDERATIONS

Unlike the current version, in the first version of our VIDA framework the Statement Composer was not designed to support joins. While using the VIDA framework that we developed, one of the first problems that we detected was the performance of the joins. This section describes the temporary solutions used to overcome this problem.

Since join syntax is usually different across database platforms, it was required to restrict its use. The in-memory join mechanism, initially developed to address in-memory data processing needs, was also used for join processing, instead of doing it at the database level.

However, this solution had evident performance problems. Making the join in the memory generates a lot of traffic in order to bring in the required data. Also, in some situations the amount of data will be too large and processing the join operation will require buffering and creation of temporary files. In case of very large tables, it will not be possible to load the entire information in the memory.

Our temporary solution was to create views for the most used joins that the application performs. From the application's point of view all the joins are seen now in the same way as regular tables. Also, the database server processes the join statements only the relevant data being fetched from the machine where the application is running.

Even if the view statements are database platform dependent, our framework still remains vendor-independent. The views are stored in the database and from the application's point of view they are schema objects just like the regular tables. Storing the views at the database level is fully decoupling them from the application.

Apart from the static views described above our framework also provides support for creating temporary views. These views are required in case a low frequency join is triggered by a user action and the amount of data for creating it is expected to be large. In such a situation, if the relevant permissions are held by the connected account, our VIDA framework will produce a view creation statement by invoking the statement composer for this purpose and will create the view as a temporary object. The application will use the temporary created view in order to get the needed joined data and the view will be dropped afterwards, when the application will exit. Of course, such an approach is inefficient for small amounts of data. In such situations it is better to perform the join directly in the memory.

The above-described methods are not anymore required in the current version of the framework, since the support for joins was added to the statement composer.

7. THE FRAMEWORK ARCHITECTURE

The architecture of the proposed framework for building VIDA is presented in Figure 4.

Our implementation was ODBC based, but support for alternate database access protocols is also provided. As shown in Figure 4, this is achieved through an Abstract Connectivity Layer used to isolate the applications from the database access protocol. The Native Translation Layer also isolates the application from the vendor specific SQL dialects, making the SQL look from the application's point of view as a well-respected standard.

The Abstract Connectivity Layer, working in conjunction with the native translation layer, provides a generic database access interface to be used by its clients (applications). The Native Translation Layer converts the SQL statements into the native SQL dialect of the connected database.

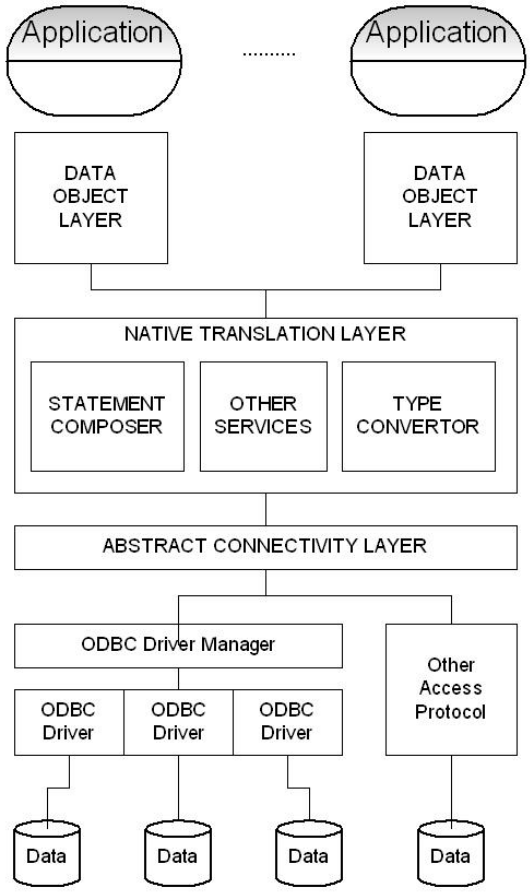

Fig.4. The Proposed Framework Architecture

Data Object Layer is the only one that is application specific, handling the object-relational conversion. Each table from the application's database has a correspondent class here, able to cope with all the required operations.

The main advantage of the proposed architecture is the separation of concerns. The object-oriented aspects of the application are separated from the relational aspects of the database and the problems of each domain can be handled using domain specific patterns.

Database tuning, locking strategies and caching are crucial to achieve acceptable performance of a business information system. [12] Usually, the tuning concentrates on the database access and is an iterative process. In such cases, tuning will affect only the access layer, leaving the application untouched.

8. PERFORMANCE MEASUREMENTS

The performance of the framework for implementing VIDA described in this paper was compared with the performance of an implementation where the database is directly accessed from the application, without passing through any isolation layer.

In order to make the test more relevant, the volume of data used was gradually increased from 1 MB to 50 MB. An Oracle 9i (9.2.0.3) was used to perform the tests.

In order to capture all the possible aspects of a database application, a mix of both DDL and DML statements was used. The test application creates a new table, alters it with some constraints, inserts a variable amount of data into it, updates one by one 1% of the records, perform various simple selects and joins with other table from the schema, and finally all the data from the newly created table is deleted and the table dropped. The application that directly accesses the database was tuned as much as possible using vendor specific optimization techniques.

As depicted in Figure 5, the overhead introduced by the isolation layers is very small. Therefore, considering its advantages, it is advisable to use the proposed architecture in order to fully decouple the application from the database.

Even though from CPU processing point of view the difference is considerable, the overall performance of the system is not impacted. This is because most of the time in a database application is spent for I/O operations (database access), the extra processing introduced by the framework being negligible.

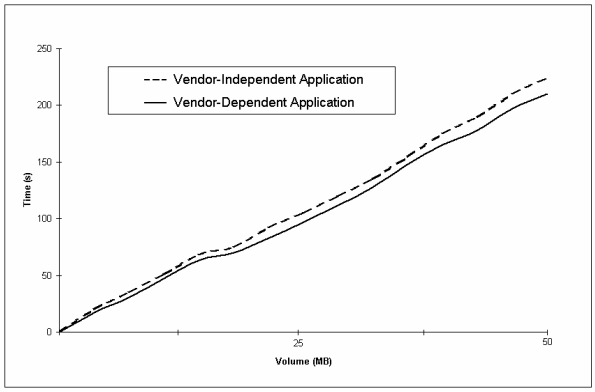

Fig.5. Performance Comparison

9. CONCLUSIONS

In this paper we presented an architecture that assures the vendor independence of a database application. Switching from one database to another having a different vendor is a challenge because no DBMS is fully compliant with the SQL standard.

Our framework enforces the SQL standard at a conceptual level. The application views SQL in a uniform way, irrespective of the vendor specific dialect in which it is converted by the Native Translation Layer.

The proposed architecture can easily accommodate new DBMS; the only component affected being the Native Translation Layer. Changing the database access protocol is also possible due to the isolation assured by the Abstract Connectivity Layer.

The ability to implement Vendor-Independent Database Applications through a well-defined and already validated framework offers important advantages like high flexibility and shorter time to market.

We used the framework presented in this paper to implement medium-scale industrial applications. The measurements showed that the processing overhead introduced by the isolation layers is not importantly affecting the overall performance of the applications. The performance trade-off is reasonable keeping in mind the important benefits in terms of database access tuning possibilities, separation of concerns and vendor independence.

Designing an aspect-oriented version of this framework is an interesting future work.

10. REFERENCES

[1] Amihai Motro, “Multiplex: A Formal Model for Multidatabases and Its Implementation”, Proceedings of Next Generation Information Technologies and Systems, 4th International Workshop, NGITS’99, pages 138 – 158, Zikhron - Yaakov, Israel, July 1999.

[2] Bertrand Meyer, Object-Oriented Software Construction (2nd Edition), Prentice Hall, 2000.

[3] Bruce Eckel, Thinking in C++ (2nd Edition), Prentice Hall, 2000.

[4] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, “Design Patters – Elements of Reusable Object-Oriented Software”, Addison-Wesley Pub. Co., 1995.

[5] Hector Garcia-Molina, Jeffrey D. Ullman and Jennifer Widom, “Database Systems – A complete book”, Prentice Hall, 2002.

[6] John Paul Ashnfelter, “Database Access Protocols”, Web Review, November 19, 1999.

[7] Mark Strawmyer, “Building Database Indepenednt Data Access”, from Codeguru’s .NET Nuts & Bolts.

[8] MSDN Library, http://msdn.microsoft.com.

[9] Octavian Paul ROTARU, Marian Dobre, Mircea Petrescu, ”A Framework for Implementing Vendor-Independent Database Applications", Proceedings of ECI'04 (6th International Scientific Conference on Electronic Computers & Informatics), Kosice - Herlany - High Tatras, Slovakia, September 22-24, 2004, pp. 68-74, ISBN 80-8073-150-0.

[10] Shelley Doll, “Is SQL a standard anymore?”, Builder.Com, June 19, 2002.

[11] Wolfgang Keller, Jens Coldewey, “Relational Database Access Layers - A Pattern Language”, Proceedings of PLoP 19996.

[12] Wolfgang Keller, “Object/Relational Access Layers – A Roadmap, Missing Links and More Patterns”, Proceedings of EuroPLoP 1998.