AN OVERVIEW OF THE OBJECT-ORIENTED DATABASE PROGRAMMING LANGUAGE DBPQL

Markus Kirchberg

Information Science Research Centre and Department of Information Systems
Massey University, Private Bag 11 222, Palmerston North 5301, New Zealand

Keywords: Database Programming Language, Object-Oriented Programming, Query Language, Impedance Mismatch, and Object Database Management System.

Abstract: In this paper, we present a new approach to the integration of object-oriented programming languages, database programming languages and query languages. While object-oriented programming languages and languages that are supported by object database systems appear to be closely related, there are a number of significant differences that affect language design and implementation. Such issues include the degree of encapsulation, persistence, the incooperation types and classes, inheritance, concurrency, NULL values, garbage collection etc. In this paper, we outline the respective challenges that affect language design and provide a brief overview of the integrated object-oriented database programming and querying language DBPQL.

1 INTRODUCTION

Traditionally, programming languages (PLs) focus on processing. In turn, data storage and data sharing only play a minor role. Long-term data is 'exported' to a file system or database system (DBS). DBSs, on the contrary, have been built to support large bodies of application programs that share data. The emergence of object-oriented PLs (OOPLs) have brought those two concepts closer together, i.e. data is at the centre of attention. Nevertheless, in OOPLs, the messages objects accept are most important while DBSs largely evolve around object persistence and data sharing. This, of course, results in a number of common (i.e. object-related) concepts being dealt with differently. Such concepts include the degree of encapsulation, treatment of transient and persistent data, incorporation of types and classes, inheritance, concurrency, NULL values, garbage collection etc.

In todays database (DB) marketplace, (object-)relational DBSs ((O)RDBSs) are dominant. Object-oriented DBSs (ODBSs) were originally thought of to replace RDBSs because of their better fit with object-oriented programming. However, high switching costs, the inclusion of object-oriented (OO) features in RDBSs, and the emergence of object-relational mappers (ORMs) have made RDBS successfully defend their dominance in the DB marketplace. ODBSs are now established as a complement, not a replacement for (O)RDBSs. Especially the open source community has created a new wave of enthusiasm that is fuelling the rapid growth of ODBS installations.

In general, DBSs provide support for query languages (QLs), most commonly SQL-like languages. It has been common practice for decades to link the PLs and DBSs domains by embedding QLs into PLs. However, the embedded approach suffers from problems collectively known as *impedance mismatch*. Alternative integrated approaches circumvent these problems. While the majority of them either represent a PL with added query constructs (e.g. DBPL (Atkinson and Buneman, 1987)) or a QL with added programming abstractions (e.g. Oracle PL/SQL (Feuerstein and Pribyl, 2002)), only a few research projects (e.g. LOQIS (Subieta, 1991)) have attempted a seamless integration of programming and querying languages. We follow the latter line of research.

In this paper, we briefly discuss issues that affect the integration of OOPLs and DB languages. Subsequently, an overview of the integrated object-oriented database programming and querying language DBPQL is presented.

2 PRELIMINARIES

In this section, we first discuss the impedance mismatch. Subsequently, attention shifts to issues that affect the integration of DB languages and OOPLs.

2.1 The Impedance Mismatch

The term *impedance mismatch* refers to an inadequate or excessive ability of one system to accommodate input from another. The *object-relational impedance mismatch* is often named a central problems of DB research. It refers to a set of conceptual and technical difficulties, which are often encountered when an (O)RDBS is being used by a program written in an OOPL. Robert Greene highlights in (Cook et al., 2006) that '*Objects in the language and Relations in the database have always been at odds, as articulated in the classic problem known as "impedance mismatch"*.' Two fundamental ways in which this mismatch materialises, i.e. as developer burden and in slower performance, are identified. While standardisation of mapping has brought relief to the former, the latter still poses a significant burden.

2.2 On the Integration of PLs and QLs

The relationship between QLs and PLs has been studied for decades. Among others, (Leontiev et al., 2002) reviews existing integrated DBPLs. The review was written as part of the TIGUKAT project (Özsu et al., 1995), which aimed at developing a novel ODBS. TIGUKAT researchers proposed a novel object model whose identifying characteristics include a purely behavioural semantics and a uniform approach to objects. However, research has been terminated without addressing problems that arise when including data creation, data manipulation and programming language constructs into a behavioural DBPL.

Another, from a practitioners point of view, more interesting approach is presented in (Subieta et al., 1993). The seamless integration of a QL with a PL is investigated. Researchers follow an extended approach to stack-based machines as known from classical PLs such as Pascal. This *Stack-Based Approach (SBA)* includes the SBQL language.

While the TIGUKAT project is based upon a powerful type system (resulting in implementation challenges and performance problems), the SBA approach rejects any type checking mechanism that originates from type theory. In addition, SBA lacks of any efforts that aim towards efficient evaluation. Concepts such as concurrent processing, transactions and distribution are neglected by both approaches altogether.

2.3 DBPLs Vs. Conventional PLs

Fundamental issues that are dealt with differently in OOPLs and in DB languages have been discussed in various papers (Bloom and Zdonik, 1987; Atkinson et al., 1989; Kim, 1993).

While OOPLs evolve around the messages an object accepts, DBSs expose both the structure and behaviour of objects. Hence, encapsulation is interpreted differently. Without relaxing the encapsulation property support for ad-hoc querying is impossible. (Atkinson et al., 1989) suggests that the support of encapsulation is essential but it may be violated under certain conditions.

The interpretation of the terms *class* and *type* largely vary even in the OOPL domain. However, we focus on corresponding differences between ODBSs and OOPLs. While types / classes commonly serve as data structuring primitive, they also serve as means of object access. Hence, types / classes must have (system-maintained) collections of objects associated. This is not the case in OOPLs, in fact, such a property is not even desired. Not only does it disable garbage collection it may also violate data abstraction. The latter is true since objects become accessible through the type / class even though they were meant to be accessible only through an abstract layer. In addition, a collection approach does not make sense for all type / class definitions since there might be no meaningful relationship between the respective objects.

Support of inheritance is an essential feature in both domains. However, there is no agreement on which inheritance types to support. Most arguments evolve around the question whether multiple implementation inheritance has its advantages. While modern OOPLs (e.g. Java and C#) chose not to support it, DBSs consider it as a necessary language feature.

Another issue concerns the life-time of objects. Persistence was always at the core of DBSs in contrast to PLs. The latter tended to rely on file systems or DBSs to maintain long-term data. This resulted in a non-uniform treatment of transient and persistent data. It is commonly accepted that persistence should be orthogonal (i.e. each object, independent of its type or class, is allowed to become persistent without explicit translation) (Atkinson and Buneman, 1987).

Further issues encompass the inclusion of NULL values and the different focus of concurrency support. The former has only recently found its way into OOPLs, i.e. into the second release of C#. The latter is commonly centred around transactions in DBSs (i.e. competition for resources) and around multithreading in PLs (i.e. cooperation).

3 A DISTRIBUTED OBJECT BASE

(Kirchberg et al., 2007) proposes the architecture of a distributed ODBS that is based on a sound theoretical framework. The lack of standard object semantics still is one of the main disadvantages of ODBSs. Research in previous decades has investigated complex values and references between data. The OO data model *OODM* from (Schewe and Thalheim, 1993) allows these different aspects to be combined. Starting from an arbitrary underlying type system a schema is defined as a set of classes, each of which combines complex values and references. In a distributed DBS, the fragmentation and allocation of an OODM schema will be still an OODM schema, but each class will be allocated to exactly one DBS node. Thus, each global object is now represented by multiple local objects. However, the structure of these local objects is still complex whereas efficient storage and retrieval requires to provide just records stored on pages. This implies to further decompose objects. The operational system adopts the basic idea of the SBA approach to implement DBS functionality. Stack-based machines are extended in a way that they execute simultaneously, support transactions and orthogonal persistence, and reflect operations on higher DBS levels.

4 OVERVIEW OF DBPQL

In a truly distributed ODBS, users are not aware of the distributed nature of the system. In fact, data independence and network and fragmentation transparency are key system properties. Thus, user requests are free of location- and communication-specific information. Only during the compilation, fragmentation, code rewriting, linking and optimisation processes such information is added.

Assume that user requests arrive in the form of DBPQL[1] programs / modules. The DBS component that processes requests consists of a DBPQL compiler, code optimisers, code rewritters (e.g. to map operations on global OODM objects to local OODM objects) etc. Here, we follow a black box approach assuming that incoming DBPQL modules are transformed into optimised evaluation plans, which are formulated in iDBPQL code.

iDBPQL code is then evaluated by a network of agents, which are realised as stack-based machines.

---

[1]DBPQL is a high-level DBPL based on the lower-level iDBPQL language. DBPQL is a modular language that supports transactions, query processing, generic requests, type, class and object creation / manipulation commands etc. We only refer to DBPQL aspects that stem from iDBPQL.
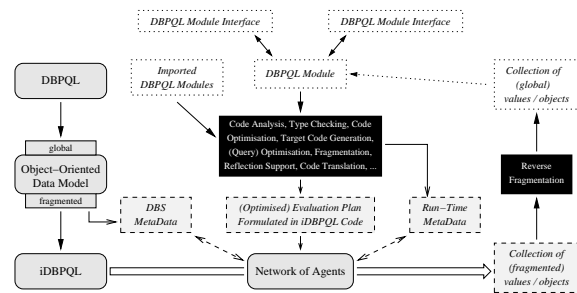

Figure 1: User Requests, Data Models and (i)DBPQL.

Agents are aware of the distributed DBS nature, know about transactions and utilise simultaneous and distributed processing. While agents process evaluation plans, they also interpret annotations that have been added by the optimiser.

Figure 1 provides a more abstract view of the relationship between DBPQL, iDBPQL, conceptual data models and associated processes. A high-level user is exposed to a few of these elements only (refer to dotted rectangles in Figure 1), which are as follows:

- A programmer codes a *DBPQL Module* and its *DBPQL Module Interface(s)*. Modules are compile-time abstractions that support the development of large-scale programs through the support of service imports and exports. Hence, information hiding is supported naturally.

- Within a module, DBPQL module interfaces of existing modules are imported. A DB schema is regarded as just another module interface with (possibly) a reduced degree of encapsulation. Following our discussions in Section 2.3, we may have regular module interfaces that strictly follow the traditional PL-interpretation of encapsulation while DB schemata expose both structure as well as behaviour. Thus, desired support for ad-hoc querying can be provided.

- Modules return results (i.e. *collections of (global) object / values*) as specified in their interface(s).

When processing DBPQL modules, code is analysed, type checked, fragmented, optimised etc. We will not consider such processes but only their result:

- DBPQL's `MAIN` method that initiates the execution is transformed into an optimised evaluation plan, the *Main Evaluation Plan*. An *Evaluation Plan* may have an *Initialisation Block*, which permits the initialisation of global and local elements before the start of the evaluation, an evaluation block and a number of associated metadata structures.

An *Evaluation Block* consists of iDBPQL statements and expressions. Concepts such as mod-

ules, interfaces, type and class definitions, code structuring primitives etc. have been removed.

- One or more schemata from the DBS metadata catalogue may be associated with the main evaluation plan. The *DBS MetaData catalogue* is a collection of compiled DB schemata. All schema imports result in such associations. Due to fragmentation, a single DBPQL import may result in a number of associated iDBPQL schemata.

- One or more run-time metadata catalogue entries are associated with each evaluation plan. The *Run-Time MetaData catalogue* is a collection of type and class definitions introduced in the user's source code or its imported modules. While DBS metadata catalogue entries describe persistent, shared data, run-time metadata catalogue entries relate to transient, non-shared data.

Evaluation plans are associated with every non-abstract behaviour specification. Classes may have static variables declared outside of any method. Such declarations are captured in the class's initialisation block. When invoking a method, the corresponding evaluation plan is executed.

Finally, we want to underline some important properties of evaluation plans. They differ from the original DBPQL modules in the following ways:

- iDBPQL code refers to DBS metadata or transient data (i.e. run-time metadata). Original code fragments have been amended in a way that references to higher-level features have either been removed or replaced by macros formulated in iDBPQL.

- A user program is translated into a main evaluation plan together with associated metadata entries. In turn, these metadata entries and references to iDBPQL library features may have further evaluation plans associated. Thus, program execution results in an evaluation of the main evaluation plan together with all evaluation plans that are encountered during its evaluation.

- Data definitions are removed from evaluation plans and now part of the metadata catalogues.

- iDBPQL statements and expressions are allocated to DBS nodes. Thus, an indication about the location of the evaluation is provided.

- Evaluation plans consist of DO ... ENDDO blocks, which are used to group statements together, form atomic execution units, model local and distributed transactions and support simultaneous processing.

- Indices and other information used to optimise processing are added (i.e. as annotations) to support the evaluation of iDBPQL statements.

5 CONCLUSION

We presented an overview of the DB programming and querying language DBPQL. This integrated language circumvents the impedance mismatch and unites properties from OOPLs, QLs and DBPLs.

REFERENCES

Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., and Zdonik, S. (1989). The object-oriented database system manifesto. In *Proceedings of the 1st International Conference on Deductive and Object-Oriented Databases*, pages 223–240, Kyoto, Japan.

Atkinson, M. P. and Buneman, P. (1987). Types and persistence in database programming languages. *ACM Computing Surveys (CSUR)*, 19(2):105–170.

Bloom, T. and Zdonik, S. B. (1987). Issues in the design of object-oriented database programming languages. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 441–451, New York, NY, USA. ACM Press.

Cook, W. R., Greene, R., Linskey, P., Meijer, E., Rugg, K., Russell, C., Walker, B., and Wittig, C. (2006). Objects and databases: State of the union in 2006. Panel at the International Conference on Object-Oriented Programming, Systems, Languages, and Applications.

Feuerstein, S. and Pribyl, B. (2002). *Oracle PL/SQL Programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Kim, W. (1993). Object-oriented database systems: Promises, reality, and future. In *Proceedings of the 19th International Conference on Very Large Data Bases*, pages 676–687, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Kirchberg, M., Schewe, K.-D., Tretiakov, A., and Wang, B. R. (2007). A multi-level architecture for distributed object bases. *Data & Knowledge Engineering*, 60(1):150–184.

Leontiev, Y., Özsu, M. T., and Szafron, D. (2002). On type systems for object-oriented database programming languages. *ACM Computing Surveys (CSUR)*, 34(4):409–449.

Özsu, M. T., Peters, R. J., Szafron, D., Irani, B., Lipka, A., and Muñoz, A. (1995). TIGUKAT: A uniform behavioral objectbase management system. *VLDB Journal*, 4(3):445–492.

Schewe, K.-D. and Thalheim, B. (1993). Fundamental concepts of object oriented databases. *Acta Cybernetica*, 11(1-2):49–84.

Subieta, K. (1991). LOQIS: The object-oriented database programming system. *Lecture Notes in Computer Science*, 504:403–421.

Subieta, K., Beeri, C., Matthes, F., and Schmidt, J. W. (1993). A stack-based approach to query languages. Technical Report 738, Institute of Computer Science Polish Academy of Sciences, Warszawa, Poland.