STUDIA INFORMATICA 2011
Volume 32 Number 3B (99)

Jarosław KOSZELA, Miłosz GÓRALCZYK, Michał JASIOROWSKI,
Marcin KARPIŃSKI, Emil WRÓBEL, Kamil ADAMOWSKI,
Joanna BRYZEK, Mariusz BUDZYN, Michał MAŁEK
Military University of Technology, Informatics System Institute

EXECUTIVE ENVIRONMENT OF DISTRIBUTED OBJECT DATABASE MUTDOD

Summary. This paper contains an overall view of main elements of an executive environment of distributed object database MUTDOD (Military University of Technology Distributed Object Database), which is designing at Military University of Technology in Warsaw.

Keywords: environment, distributed object database, MUTDOD

ŚRODOWISKO WYKONAWCZE ROZPROSZONEJ OBIEKTOWEJ BAZY DANYCH

Streszczenie. Materiał zawiera ogólny zarys głównych mechanizmów środowiska wykonawczego rozproszonej, obiektowej bazy danych MUTDOD (ang. *Military University of Technology Distributed Object Database*), która jest tworzona w Wojskowej Akademii Technicznej w Warszawie.

Słowa kluczowe: środowisko, rozproszona obiektowa baza danych, MUTDOD

1. Introduction to MUTDOD's architecture

MUTDOD stands for Military University of Technology Distributed Object Database, which is being created at Military University of Technology. MUTDOD is meant to be a next step in a database engine evolution - it pretends to be a platform for object systems containing applications coded in object languages (e.g. C#, Java, ect.), which are working with data stored in object database.

There are some advantages of MUTDOD. First of all it is a pure objective environment for all elements in the object system. Unified data model for applications and database causes

that there is no need to use any ORM (Object-Relational Mapping). What is more, programmers do not have to use special connectors to get data from a database, but they can just query for data with an easy and intuitional syntax – similar to one used to access application's local objects stored in memory.

A second advantage of MUTDOD is its distribution environment. Nowadays it is nothing special to work in multi-core and multi-host environment, but unfortunately most applications are using only one core. "Common distribution" required an additional effort from a programmer, who has to divide manually algorithms into parts. MUTDOD is meant to help the programmer handle this problem by providing a distributed database environment which is as easy to use as a single node one.

At this point many cases of a MUTDOD's inner processing are considered to be developed, some of them are already designed and a few of them are already implemented. This paper provides an overall description of main mechanisms of MUTDOD execution environment. Authors of this article put an effort to describe most interesting cases of their work.

2. Object-oriented language

The query language is a very important aspect of MUTDOD project. MUTDOD`s semantic, syntax and pragmatics include new ideas, but it still stays user-friendly and enables easy search using various of criteria. Query language, designed for MUTDOD purposes, contains elements from both declarative and imperative languages. Declarative statements are independent and isolated, so they could be executed independently. Scattering all calculations is much easier because of that.

Table 1

Simple query in SQL and DODQL

SQL	DODQL[1]
select e.Name, e.Surname from Employees where e. profession = 'director';	Employees.Director(Name, Surname);

Requests should be generated in a way, which allows token`s tree optimization. This provides ability to calculate result of queries not only on one machine, but also on multiple computers. To perform multicomputer calculations, the base form of token tree has to be divided into separate smaller unrelated queries, which will then be executed in parallel. The query language stays consistent even when some objectivity paradigms are added to it. It will allow

[1] DODQL – a MUTDOD query language, based on SBQL (Stack Based Query Language); see more at www.sbql.pl

user to intuitively use the new language, without the need of learning syntax from the beginning. For example, if we would like to get a name and surname of company`s director, we could write in the way presented in table 1.

This is much shorter than the same query written in another language too. Thanks to that, programmers don't have to focus on syntax, because it is clear and intuitive, but on the other hand which syntax of a sentence, from the following example, will be more intuitive for programmers? The one, where the object from which data will be taken is on the beginning, on the end or maybe on some place in the middle of the query:

```
Univesity.Department.Dean(Degree, Name, Surname);
(Degree, Name, Surname)Dean.Departmant.University;
(Degree, Name, Surname) Univesity.Department.Dean;
```

Nowadays some query languages pretend to be objected-oriented e.g. OQL, LINQ, but still they base on relational databases and transform works on object to queries. They generate queries from objects and return objects as result, but still all operations are executed on tables in database. In the following example, we can compare how the same query in OQL, LINQ, could look at MUTDOD language:

Table 2

Simple query in OQL, LINQ and DODQL

OQL	LINQ	DODQL
select struct (E: e.name, :e.dept.name) from e in Employees as e where e.id='10'	from e in db.Employees where e.Id = 10 select new{ e.Name, e.DeptName}	(name,deptName) Employees(where Id = 10)

MUTDOD is going to work without query languages as SQL, OQL or LINQ. Query language and database model will be object-oriented. Thanks to that we can select and union different objects. When user`s query asks for an object, which does not exist in the database, do not have its own class, then the generated temporary object is generated, which exist as long as the session exist, e.g. *average salary*.

A lot of different issues have to be solved in MUTDOD, because syntax, semantics and pragmatics of object-oriented language is the most issue. Even the best solutions won't be adopted unless they are clear, easy and user-friendly.

3. Storing of objects

Core part of every database management system is data storage. There are many problems, concerning storing objective data, which MUTDOD design need to consider. Some of them are related to physical layer (close to hardware) and I/O operations optimization, others

are correlated with logical data structures which need to be organized in such a way that smallest possible number of mappings is needed to retrieve objects based on their *OID (Object Identifier)*. The storage also needs to keep track of objects and metadata stored in one instance of database engine which ensures consistency of data. We also have to resolve problems concerning efficient serialization and deserialization of objects so that they can be stored in form of binary data in persistent data store. Another important requirement of data storage is the ability to access every single object's property without activating (creating an instance of) the object. This is needed for performing fast queries on object's properties which are not indexed.

Because read/write operations are often performed on the database in a short time in a context to the same data piece (the same object in this case) so that the cache mechanism is a very important part of storage which speeds up this kind of operations. This reduces costly persistent storage I/O operations.

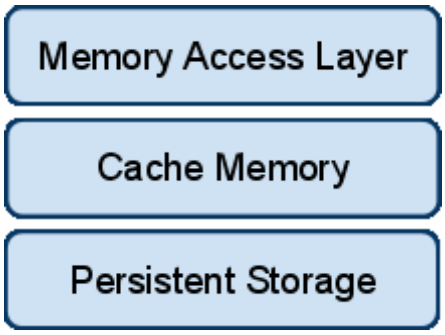

Fig. 1. Data access hierarchy diagram
Rys. 1. Diagram hierarchii danych

After facing these problems during design and implementation of MUTDOD we ended up with the solution to persist the objective data in such a way that we can easily and efficiently access the complex data structures represented by the objects stored in the objective data store.

4. Indexes

Object oriented environment requires a few different types of indexes for base types and user objects. Usually an object consists of other objects and that must be reflected in local indexes of object. Double way inclusion is easy to delete from indexes when one object is being disposed. Problem comes with one way inclusion. After destruction of object contained in the other object, all indexes for this object should be deleted. This requires iterating through all local indexes of all objects in system. Of course this is very absorbing for system and may lock local index for a while. One of ideas to manage with one way inclusion is not to

delete indexes after deleting of object, but after first attempt to access not existing object. The result is growing size of index. An idea to prevent that is to iterate through all objects and local indexes from time to time and remove unnecessary data. Another problem is after casting object to another type. Object after casting should lose local index in other object (one way inclusion) but both objects should be still alive, in order to keep indexes consistent, all local indexes should be checked. For that situation, index cannot be deleted after first attempt, because both objects are still in system and OID (**O**bject **Id**entifier) is static. Solution for all problems with one way inclusion is collecting indexes in both ways. That simplifies operations on indexes after changing states of objects, but increases size of indexes.

Another type of index is global OID based index of all objects in database. This kind of index is required to quickly find place in memory where object is stored. Key of index should be created in a way that helps comparing objects. For example if system gets query, with OID, to look for similar objects, after looking in indexes, system should get set of objects to compare with details. That would reduce number of elements to compare, discard objects that are surely not similar after comparing indexes keys and decrease time of query execution. Global OID index should also contain information about server where data is stored for multimode database system. This information should be used for locating files, but is also useful for mechanism of dividing execution of query between computers in database system.

Extension index is typical object-oriented architecture index. Indexing mechanism should contain two types of this index: simple and compound. First type requires to collect all objects in database of the same class. Compound index is going to get together all objects of the same class and objects which get inherit from this class. In fact that will be the sum of all objects in simple index for inheritance tree. Dynamic inheritance delivers a lot of operations so that extension indexes should be very fast and react immediately on object change. This kind of index should be coherent in any time.

MUTDOD system is going to provide a few kinds of indexes. In fact most important indexes for user will be user-detained indexes. System needs to have complete system of indexes, quick and providing consistent state. Testing different ideas and algorithms for storing and operating on indexes provides some schemas for the best balance between size and number of operations on indexes after typical changing objects states.

5. Query Execution

DOD's architecture provides two types of servers: central machine and data machine. Central machine is a logic server. It manages other machines and supports pre-executing queries. If the central server stops working the other machines choose a new one. Data machines

store objects and executing queries provided by central machine. More information about MUTDOD's server architecture can be found in related works [Object-Oriented Distribution in MUTDOD].

Executing queries in MUTDOD system must be managed and optimized for a distributed environment. Incoming query goes to actual central machine or to node the operator has chosen (in full p2p mode). The first step is analysis of the query and checking which machine has the most free resources to resolve it. Other thing during choosing machine is checking how many of the objects needed to resolve query are stored on it. Query analysis checks if it is possible to divide the query execution to several machines. When it's possible, query is divided and sent to selected machines with information which machine should get results of executing this part of the query. The machine that manages executing process wait for all parts of the query result and combines it into a complete result that can be send back to user. In a case when query must be execute in one machine it's send to it with information how to send the result to the user.

Integration of the query language with a programming language allows using the full potential of object-oriented database including the absence of non-impedance. Moving query interpretation and optimization module to client side decreases usage of server resources and allows server to handle more clients. This solution requires storing some metadata on client-side. It needs metamodel, available procedures and functions.

6. Cache

Caching is one of basic mechanisms, which leads to a query result retrieval time optimization. It has a key impact on a database system performance. An immediate answer for the same re-executed query is not the main mechanism's function. It shortens evaluation time of a query that has a little similarity to any one which already has been evaluated. Token`s tree is divided into independent parts. Each part is stored separately. That solution requires bigger buffer size but gives extra opportunity. Here comes an example:

Finding independent subqueries is a way to increase probability of storing required result (and avoiding data redundancy), but it is not the only one. The buffer sooner or later gets full. Mechanism deletes some of the result to make space for new ones. Which are the less valuable ones is a very important question. Algorithm of data value estimation is crucial for attaining best buffer hit ratio. There are many approaches to the issue: LRU (based on last data reference), NFU (based on number of data reference), Aging (mixed) and others. All of these approaches will be available in MUTDOD. The mechanism will be able to adaptation. It will

work for a period of time, gather statistics and choose the best algorithm for the present environment.

Division of query into subqueries, using indices, removing dead subquery, removing auxiliary names, pushing selection before join are domains of other caching mechanisms. Applying them all will greatly increase database performance.

Table 3

Caching process steps

Original query	Employee(where Name = “John” && Seniority = “10” && Superior = “Marvin”).Salary
Original query is evaluated and stored as three separated subqueries	Employee(where Name = “John”).Salary Employee(where Seniority = “10”).Salary Employee(where Superior = “Marvin”).Salary
That result in giving immediate answer for following query	Employee(where Seniority = “10” && Superior = “Marvin”).Salary
Caching mechanism returns intersection of suitable already stored query results. In addition following query evaluation time	Employee (where Employee.Name = “John” && Employee. Seniority = “10” && Employee.Superior = “Marvin” && Employee.Occupation = “accountant”). Salary
Is the same as this subquery	Employee.Occupation = “accountant”

7. Client environment

Problem of integrating programming languages and databases remains open, since database systems came into common use. Despite many studies on this subject, a widely accepted combination of these elements still has not been developed. The most important reason for that may be the lack of standardization and fragmentation of the current solutions.

First attempts to define the problem in the field of object database integration may be found in works of Carey and DeWitt [1], where the client integration is described as one of key elements of further researches and ODMG Language Bindings presented in their Specification [2].

Integration of common programming languages (such as Java or C#) and database query languages is complex, because they are based on different semantic foundations. Imperative expressions describe the steps needed to be made to get the proper product, when declarative queries emphasize the result over the way to reach it.

In most cases direct data mapping is impossible due to differences in primitive types and object definitions. Furthermore, database providers may have different look at some topics

including encapsulation, multiple inheritance or methods persistence, which remain open according to "The object-oriented database system manifesto" [3].

Related works [4] highlight aspects like compile time query optimization, foster programmers habits and IDEs integration, as a key to popularization of specific solution.

There are two main types of client integration:

- **Explicit Query Execution** – which can be realized both by provided APIs, where all database operations are hidden behind interface unifying data access or *Embedded Query Languages* (LINQ, JSQL). Second method is much safer and easy for use because it is not based on string expressions that cannot be checked, but makes language more complex.
- **Orthogonal Persistence** - every object is persistent during its lifetime. There is no need for any special actions to save its state. Since in most popular scenarios not all objects have to be stored for further use *Degree Persistence* is used more often, where explicit transaction calls are required to store object's state.

In accordance with the adopted model of a database, SBQL was chosen to be a query language. Main task was to create MUTDOD connectivity wrapper providing easy query call mechanism. It was developed as an API level access mechanism allowing string based queries execution. First implementation was created to be called from C# 4.0 language, with usage of its late binding features. It was supporting basic types mapping and object nested structure access in C# like way, with method calls possibility.

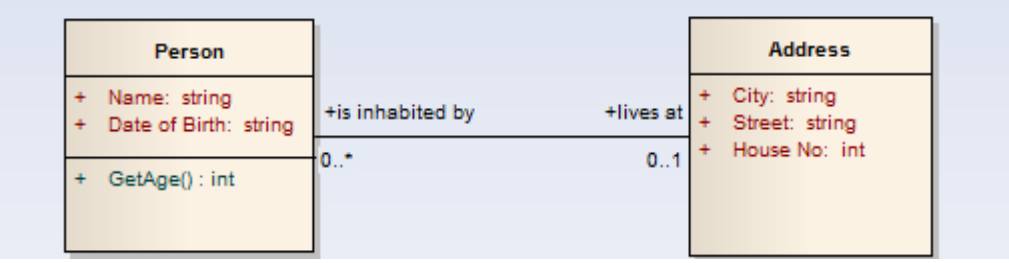

Fig. 2. Simple database class diagram
Rys. 2. Prosty diagram klas bazy danych

Currently the main objective is the creation of SBQL based embedded query language. It should have full language integration with basic types mapping, and database objects call possibility from native code. We assumed no need for database model mapping before its use and no object mapping because of possible metamodel differences. Additional works are carried out to provide IDE support including auto-completion, query check and precompilation optimization.

8. Summary

As a research project Distributed Object Database System is constantly evolving. Described solutions, mechanisms and thesis are constantly verified and modified, not rarely new ones are developed. Final look of MUTDOD is yet not specified. This article should show only the basis and direction of MUTDOD's evolution. Described problems definitely show that design and implementation of such complex environment as object-oriented database with ability to distribute both data storing and query execution is not an easy task. Provided aspects show that MUTDOD system design is based on usability and comfort of its users. Every module of MUTDOD conforms to already mentioned characteristics but also security and performance are not forgotten. Fully developed MUTDOD has huge chances to become a widely used object systems platform.

BIBLIOGRAPHY

Carey M. J., Dewitt D. J.: Of Objects and Databases: A Decade of Turmoil. http://www-.cs.ubc.ca/~rap/teaching/504/2005/readings/objects.pdf, downloaded at 31th January 2011.

1. Barry D.: ODMG 2.0: A Standard for Object Storage. http://www.inf.puc-rio.br/ /~casanova/INF1731-BD/Referencias/odmg20-storage.pdf, downloaded at 31th January 2011.
2. Atkinson M., Bancilhon F., DeWitt D., Dittrich K., Maier D., Zdonik S.: The Object-Oriented Database System Manifesto. http://reference.kfupm.edu.sa/content/o/b/ /the_object_oriented_database_system_mani_85098.pdf, downloaded at 31th January 2011.
3. Cook W. R., Rosenberger C.: Native Queries for Persistent Objects. A Design White Paper. http://www.odbms.org/download/010.01%20Cook%20Native%20Queries%20for %20Persistent%20Objects%20August%202005.pdf, downloaded at 31th January 2011.
4. Catell R. G. G.: The Object Database Standard: ODMG 3.0. Morgan Kaufmann, 2000.
5. Subieta K.: Teoria i konstrukcja obiektowych języków zapytań. Wydawnictwo PJWSTK, Warszawa 2004.
6. Lausen G., Vossen G.: Obiektowe bazy danych Modele danych i języki. Warszawa 2000.
7. Brzozowska P., Góralczyk M., Jesionek Ł., Karpiński M., Kędzierski G., Kędzierski P., Koszela J., Wróbel E.: System obiektowy = obiektowa baza danych + obiektowa aplikacja. Studia Informatica, Vol. 31, No. 2B (90), Gliwice 2010.
8. Góralczyk M.: Projekt oprogramowania zarządzającego obiektową bazą danych. Praca dyplomowa WAT, Warszawa 2010.

9. Karpiński M.: Projekt mechanizmu replikacji i synchronizacji elementów obiektowej bazy danych. Praca dyplomowa WAT, Warszawa 2010.
10. Brzozowska P.: Projekt analizatora syntaktycznego i semantycznego obiektowego języka zapytań. Praca dyplomowa WAT, Warszawa 2010.
11. Wróbel E.: Projekt interfejsu programistycznego dostępu do obiektowej bazy danych. Praca dyplomowa WAT, Warszawa 2010.
12. Connoly T., Begg C.: Database Systems: A Practical Approach to Design, Implementation, and Management. Addison-Wesley, 2002.

Recenzenci: Dr inż. Łukasz Wyciślik
Dr inż. Hafed Zghidi

Wpłynęło do Redakcji 17 stycznia 2011 r.

Omówienie

MUTDOD to Military University of Technology Distributed Object Database, czyli rozproszona, obiektowa baza danych tworzona w Wojskowej Akademii Technicznej. MUTDOD jest tworzony jako platforma dla systemów tworzonych w obiektowych językach programowania.

Podstawowym elementem MUTDOD jest zaprojektowany od podstaw nowy język programowania, łączący zalety zarówno języków deklaratywnych, jak i imperatywnych. Język taki musi być przede wszystkim bardzo pragmatyczny i wprowadzać jak najmniej elementów nieznanych do tej pory programistom. Ponadto, nowy język będzie musiał sprostać zadaniu manipulowania nie tylko danymi, ale także całym środowiskiem wykonawczym.

Istotnym problemem, z którym twórcy MUTDOD-a musieli się zmierzyć, było też wydajne zachowywanie danych na nośnikach komputerów. Głównym problem okazał się tutaj taki dostęp do obiektów, który nie wymagałby powoływania ich instancji.

Również mechanizm indeksowania wymagał gruntownego przemyślenia. Baza, w związku z możliwością pracy rozproszonej, wymaga przechowywania nie tylko indeksów lokalnych, ale również indeksu globalnego, zawierającego rozmieszenie obiektów na węzłach. Istotne dla indeksów było również znalezienie idealnego mechanizmu rozwiązywania problemów z powiązaniami jednokierunkowymi.

Wiele pracy w systemie MUTDOD zostało poświęcone rozproszonemu wykonywaniu zapytań. System musi sobie radzić z podziałem zadań na mniejsze elementy oraz z łączeniem wyników podzapytań w wynik ostateczny, stanowiący odpowiedź na zapytanie użytkownika.

Addresses

Jarosław KOSZELA: Wojskowa Akademia Techniczna, Wydział Cybernetyki,
ul. gen. Sylwestra Kaliskiego 2, 00-908 Warszawa, Polska, jkoszela@wat.edu.pl.
Miłosz GÓRALCZYK: Wojskowa Akademia Techniczna, Wydział Cybernetyki,
ul. gen. Sylwestra Kaliskiego 2, 00-908 Warszawa, Polska, milosz.goralczyk@gmail.com.
Michał JASIOROWSKI: Wojskowa Akademia Techniczna, Wydział Cybernetyki,
ul. gen. Sylwestra Kaliskiego 2, 00-908 Warszawa, Polska, michajas@gmail.com.
Marcin KARPIŃSKI: Wojskowa Akademia Techniczna im. Jarosława Dąbrowskiego,
ul. gen. Sylwestra Kaliskiego 2, 00-908 Warszawa, Polska, marcin@karpinski.waw.pl.
Emil WRÓBEL: Wojskowa Akademia Techniczna im. Jarosława Dąbrowskiego,
ul. gen. Sylwestra Kaliskiego 2, 00-908 Warszawa, Polska, wrobel.emil@gmail.com.
Kamil ADAMOWSKI: Wojskowa Akademia Techniczna im. Jarosława Dąbrowskiego,
ul. gen. Sylwestra Kaliskiego 2, 00-908 Warszawa, Polska, kamiladamowski@gmail.com.
Joanna BRYZEK: Wojskowa Akademia Techniczna im. Jarosława Dąbrowskiego,
ul. gen. Sylwestra Kaliskiego 2, 00-908 Warszawa, Polska, joannabryzek@gmail.com.
Mariusz BUDZYN: Wojskowa Akademia Techniczna im. Jarosława Dąbrowskiego,
ul. gen. Sylwestra Kaliskiego 2, 00-908 Warszawa, Polska, mariuszbudzyn@gmail.com.
Michał MAŁEK: Wojskowa Akademia Techniczna im. Jarosława Dąbrowskiego,
ul. gen. Sylwestra Kaliskiego 2, 00-908 Warszawa, Polska, malek_michal@jabster.pl.