Object-Oriented Wrapper for Relational Databases in the Data Grid Architecture

Kamil Kuliberda[1], Jacek Wislicki[1], Radoslaw Adamus[1], Kazimierz Subieta[1,2,3]

[1] Computer Engineering Department, Technical University of Lodz, Lodz, Poland
[2]Institute of Computer Science PAS, Warsaw, Poland
[3]Polish-Japanese Institute of Information Technology, Warsaw, Poland
[kkulibe, jacenty, radamus]@kis.p.lodz.pl, subieta@pjwstk.edu.pl

Abstract: The paper presents a solution of the problem of wrapping relational databases to an object-oriented business model in the data grid architecture. The main problem with this kind of wrappers is how to utilize the native SQL query optimizer, which in majority of RDBMS is transparent for the users. In our solution we use the stack-based approach to query languages, its query language SBQL, updateable object-oriented virtual views and the query modification technique. The architecture rewrites the front-end OO query to a semantically equivalent back-end query addressing the M0 object model that is 1:1 compatible with the relational model. Then, in the resulting SBQL query the wrapper looks for patterns that correspond to optimizable SQL queries. Such patterns are then substituted by dynamic SQL *execute immediately* statements. The method is illustrated by a sufficiently sophisticated example. The method is currently being implemented within the prototype OO server ODRA devoted to Web and grid applications.

1 Introduction

The art of object-oriented wrappers build on top of relational database systems has been developed for years – first papers on the topic are dated to late 80-ties and were devoted to federated databases. The motivation for the wrappers is reducing the technical and cultural difference between traditional relational databases and novel technologies based on object-oriented paradigms, including analysis and design methodologies (e.g. based on UML), object-oriented programming languages (C++, Java, C#, and others), object-oriented middleware (e.g. based on CORBA), object-relational databases and pure object-oriented databases. Recently, Web technologies based on XML/RDF also require similar wrappers. Despite the big pressure on object-oriented and XML-oriented technologies, people are quite happy with relational databases and there is a little probability that the market will massively change soon to other data store paradigms.

Unfortunately, the object-orientedness has as many faces as existing systems, languages and technologies. Thus, the number of combinations of object-oriented options with relational systems and applications is very large. Additionally, wrappers can have different properties, in particular, can be proprietary to applications or generic, can deal with updates or be read-only, can materialize objects on the wrapper

side or deliver purely virtual objects, can deal with object-oriented query language or provide some iterative “one-object-in-a-time” API, etc [1]. This causes an extremely huge number of various ideas and technologies. For instance, Google reports more than 100 000 Web pages as a response to the query “object relational wrapper”.

In this paper we deal with object-to-relational wrappers for distributed, heterogeneous and redundant data and service resources that are to be virtually integrated into a centralized, homogeneous and non-redundant whole. The technology is recently referred to as a “data-intensive grid” or a ”data grid”. While originally the grid technology denotes massive computations that have to be done in parallel on hundreds or thousands of small computers, in business applications a data grid means higher forms of distribution transparency plus some common infrastructures build on top of the grid, including the trust infrastructure (security, privacy, licensing, payments, etc.), web services, distributed transactions, workflow management, etc [2].

The major problem with the described architecture concerns how to utilize an SQL optimizer. In all known RDBMS-s the optimizer and its particular structures (e.g. indices) are transparent to the SQL users. A naive implementation of the wrapper causes that it generates primitive SQL queries such as *select * from R*, and then, processes the results of such queries by SQL cursors. Hence the SQL optimizer has no chances to work. Our experience has shown that direct translation of object-oriented queries into SQL is unfeasible for a sufficiently general case.

The solution to this problem presented in this paper is based on the object-oriented query language SBQL, virtual object-oriented views defined in SBQL, query modification [13], and an architecture that will be able to detect in a query syntactic tree some patterns that can be directly mapped as optimizable SQL queries. The patterns match typical optimization methods that are used by the SQL query optimizer, in particular, indices and fast joins. The idea is currently being implemented within our object-oriented platform ODRA.

The rest of the paper is organized as follows. In Section 2 we present a more detailed discussion concerning object-oriented wrappers built on top of relational databases, including our experience. Section 3 shortly introduces the Stack-Based Approach (SBA) to object-oriented query languages, its query language SBQL and virtual updateable object-oriented views. The section presents only basic ideas - the approach has already resulted in extensive literature (e.g. [12]) and several implementations. Section 4 presents the data grid architecture. Section 5 discusses an object-relational wrapper and presents a simple example showing how it works. Section 6 concludes.

2 More Discussion of the Problem

Mapping between a relational database and a target global object-oriented database should not involve materialization of objects on the global side, i.e. objects delivered by such a wrapper should be virtual. Materialization is simple, but leads to many problems, such as storage capacity, network traffic overhead, synchronization of global objects after updates on local servers, and (for some applications) synchronization of local servers after updates of global objects. Materialization can also be forbidden by security and privacy regulations.

If global objects have to be virtual, they are to be processed by a query language and the wrapper has to be generic, we are coming to concept of virtual object-oriented database views that do the mapping from tables into objects. Till now, however, sufficiently powerful object-oriented views are still a dream, despite a lot of papers and some implementations. The ODMG standard does not even mention views[1]. The SQL-99 standard deals a lot with views, but currently it is perceived as a huge set of loose recommendations rather than as entirely implementable artifact. In our opinion, the Stack-Based Approach and its query language SBQL offer the first and universal solution to the problem of updateable object-oriented database views. In this paper we show that the query language and its view capability can be efficiently used to build optimized object-oriented wrappers on top of relational databases.

Basing on the knowledge and experience[2] gained from our previous attempts to wrap relational content into its object-oriented representation, currently we are implementing (under .NET) an object-oriented platform named ODRA for Web and grid applications, thus the problem of a wrapper on top of relational databases comes again into the play. After previous experience we have made the following conclusions:

- the system will be based on our own, already implemented, object-oriented query language SBQL (described shortly in Section 3), which has many advantages over OQL, XQuery, SQL-99 and other languages,
- the system will be equipped with a powerful mechanism of object-oriented virtual updateable views based on SBQL. Our views have the power of algorithmic programming languages, hence are much more powerful than e.g. SQL views. A partial implementation of SBQL views is ready too [7].

The architecture assumes that a relational database will be seen as a simple object-oriented database, where each tuple of a relation is mapped virtually to a primitive object. Then, on such a database we define object-oriented views that convert such primitive virtual objects into complex, hierarchical virtual objects conforming to the global canonical schema, perhaps with complex repeated attributes and virtual links among the objects. Because SBQL views are algorithmically complete, we are sure that every such a mapping can be expressed. Moreover, because SBQL views can possess a state, have side effects and be connected to classes, one would be able to implement a behavior related to the objects on the SBQL side.

The major problem concerns how to utilize the SQL optimizer. After our previous experience we have concluded that static (compile time) mapping of SBQL queries into SQL is unfeasible. On the other hand, a naive implementation of the wrapper, as presented above, leaves no chances to the SQL optimizer. Hence we must use optimizable SQL queries on the back-end of the wrapper.

The solution of this problem is presented in this paper. It combines SBQL query engine with the SQL query engine. There are a lot of various methods used by an SQL optimizer, but we can focus on three major ones: *rewriting* (e.g. pushing selections before joins), *indices* (i.e. internal auxiliary structures for a fast access), *fast joins* (e.g. hash joins).

---

1 The *define* clause of OQL is claimed to be a view, but this is misunderstanding: it is a macro-definition (a textual shorthand) on the client-side, while views are server-side entities.

2 A gateway from the DBPL system to Ingres and Oracle (1993) and a part of the European project ICONS (Intelligent COntent maNagement System), IST-2001-32429

Concerning rewriting, our methods are perhaps as good as SQL ones, thus this kind of optimization will be done on the SBQL side. Two next optimizations cannot be done on the SBQL side. The idea is that an SBQL syntactic query tree is first modified by views [13], thus we obtain a much larger tree, but addressing a primitive object database that is 1:1 mapping of the corresponding relational databases. Then, in the resulting tree we are looking for some patterns that can be mapped to SQL and which enforce SQL to use its optimization method. For instance, if we know that the relational database has an index for Names of Persons, we are looking in the tree the sub-trees representing the SBQL query such as:

```
Person where Name = "Doe"
```

After finding such a pattern we substitute it by the dynamic SQL statement:

```
exec_immediately(select * from Person where Name = "Doe")
```

enforcing SQL to use the index. The result returned by the statement is converted to the SBQL format. Similarly for other optimization cases. In effect, we do not require that the entire SBQL query syntactic is to be translated to SQL. We interpret the tree as usual by the SBQL engine, with except of some places, where instead of some subtrees we issue SQL *execute immediately* statements.

3 Stack Based Approach, SBQL and Updatable Object Views

In the stack-based approach (SBA) a query language is considered a special kind of a programming language. Thus, the semantics of queries is based on mechanisms well known from programming languages like the environment stack. SBA extends this concept for the case of query operators (selection, projection/navigation, join, quantifiers, etc.). Using SBA, one is able to determine precisely the operational semantics (abstract implementation) of query languages, including relationships with object-oriented concepts, embedding queries into imperative constructs, and embedding queries into programming abstractions: procedures, functional procedures, views, methods, modules, etc.

SBA is defined for a general object store model. Because various object models introduce a lot of incompatible notions, SBA assumes some families of object store models which are enumerated M0, M1, M2 and M3. The simplest is M0, which covers relational, nested-relational and XML-oriented databases. M0 assumes hierarchical objects with no limitations concerning nesting of objects and collections. M0 covers also binary links (relationships) between objects. Higher-level store models introduce classes and static inheritance (M1), object roles and dynamic inheritance (M2), and encapsulation (M3). For these models there have been defined and implemented the query language SBQL, which is much more powerful than ODMG OQL [10] and XML-oriented query languages such as XQuery [14]. SBQL, together with imperative extensions and abstractions, has the computational power of programming languages, similarly to Oracle PL/SQL or SQL-99.

Rigorous formal semantics implied by SBA creates a very high potential for the query optimization. Several optimization methods have been developed and implemented, in particular methods based on query rewriting, indices, removing dead queries, and others [11].

SBQL is based on the principle of compositionality, i.e. semantics of a complex query is recursively built from semantics of its components. In SBQL, each binary

operator is either algebraic or non-algebraic. Examples of algebraic operators are numerical and string operators and comparisons, aggregate functions, union, etc. Examples of non-algebraic operators are selection (where), projection/navigation (the dot), join, quantifiers (∃, ∀), and transitive closures. The semantics of non-algebraic operators is based on a classical environmental stack, thus the name of the approach.

The idea of SBQL updatable views relies in augmenting the definition of a view with the information on user intentions with respect to updating operations. The first part of the definition of a view is the function, which maps stored objects onto virtual objects (similarly to SQL), while the second part contains redefinitions of generic operations on virtual objects. The definition of a view usually contains definitions of subviews, which are defined by the same principle [4].

The first part of the definition of a view has the form of a functional procedure. It returns entities called *seeds* that unambiguously identify virtual objects (usually seeds are OIDs of stored objects). Seeds are then (implicitly) passed as parameters of procedures that overload operations on virtual objects. These operations are determined in the second part of the definition of the view. There are distinguished several generic operations that can be performed on virtual objects: *delete* removes the given virtual object, *retrieve* (dereference) returns the value of the given virtual object, *navigate* navigates according to the given virtual pointer, *update* modifies the value of the given virtual object according to a parameter, etc.

All procedures, including the function supplying seeds of virtual objects are defined in SBQL and can be arbitrarily complex [4].

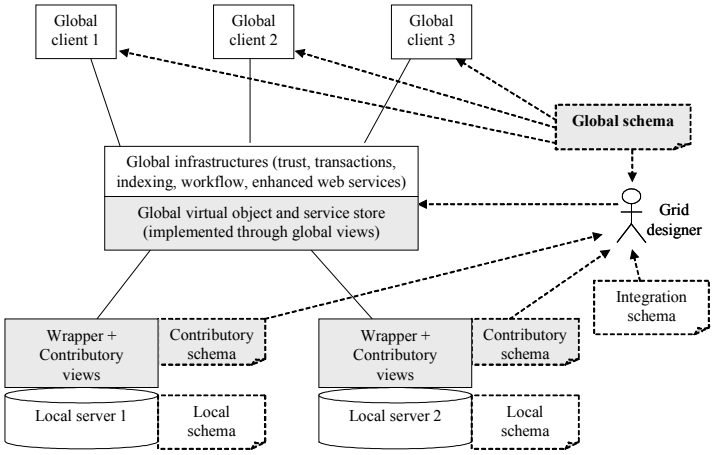

Fig. 1. Architecture of a data grid

4 Architecture of the Data Grid

Figure 1 shows the architecture of a data grid. Its central part is the *global virtual store* containing virtual objects and services. Its role is to store addresses of local servers and to process queries sent from *global client* applications. The global virtual store presents the business objects and services according to the *global schema*, which

has to be defined and agreed upon the organization creating the grid. The global schema is used by programmers to create global client applications. The grid integrates services and objects physically stored in the *local servers*. Administrators of local servers define *contributory schemata* and corresponding *contributory views* [3, 5], mapping local data and services to the global schema demands. Local data can be stored within any kind of DBMS providing a corresponding wrapper plus contributory views are implemented. The *global virtual store* is a collection of views that are responsible for the integration of distributed, heterogeneous and redundant resources and ensure higher-level transparencies. The contributory views and global views are updatable. The *integration schema* presents information on dependencies between local servers (replications, redundancies, etc.) [3, 6].

5 Architecture of the Object-Relational Wrapper and Examples

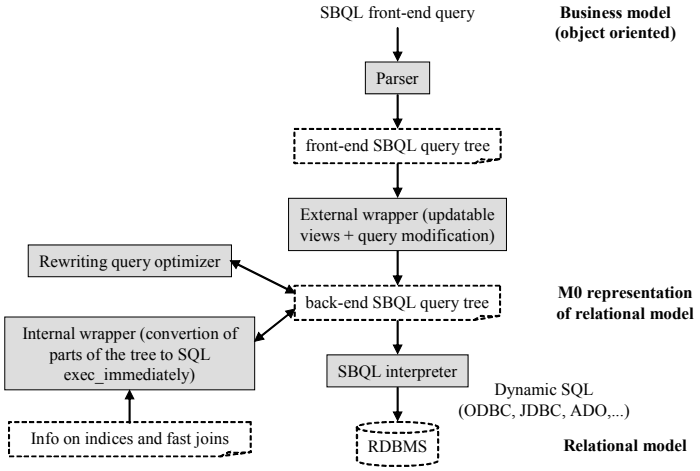

Fig. 2. The architecture of a generic wrapper for relational databases

Figure 2 presents the architecture of the wrapper. The general assumptions are the following:

- externally the data are designed according to the OO model and the business intention of the *global schema* – the *front-end* of the wrapper (SBQL),
- internally the relational structures are presented in the M0 model (excluding pointers and nesting levels above 2) [12] – the *back-end* of the wrapper (SBQL),
- the mappings between *front-end* and *back-end* is defined with updatable object views. They role is to map *back-end* into *front-end* for querying and *front-end* onto *back-end* for updating (virtual objects),
- for global queries, if some not very strict conditions are satisfied, the mapping form front-end into back-end query trees is done through query modification, i.e macro-substituting every view invocations in a query by the view body.

5.1 Updates Through the Wrapper and the Optimization Procedure

The presented architecture assumes retrieval operations only, because the query modification technique assumed in this architecture does not work for updates. However, the situation is not hopeless (although more challenging). Because in SBQL updates are parameterized by queries, the major optimizations concern just these parameters, with the use of the query modification technique as well. Then, after the optimization, we can develop algorithms that would recognize in the back-end query tree all updating operations and then, would attempt to change them to dynamic SQL *update*, *delete* and *insert* statements. There are technical problems with identification of relational tuple within the SBQL engine (and further in SQL). Not all relational systems support *tuple identifiers* (tids). If tids are not supported, the developers of a wrappers must relay on a combination (*relation_name, primary_key_value(s)*), which is much more complicated in implementation. Tids (supported by SQL) simply and completely solve the problem of any kind of updates.

In Figure 2 we have assumed that the internal wrapper utilizes information on indices and fast joins (primary-foreign key dependencies) available in the given RDBMS. In cases of some RDBMS (e.g. MS SQL Server) this information cannot be derived from the catalogs. Then, the developers are forced to provide an utility allowing the wrapper designer to introduce this information manually.

The query optimization procedure (looking from wrapper's front-end to back-end) for the proposed solution can be divided into several steps:

1. Query modification is applied to all view invocations in a query, which are macro-substituted with seed definitions of the views. If an invocation is preceded by the dereference operator, instead of the seed definition, the corresponding *on_retrieve* function is used (analogically, *on_navigate* for virtual pointers). The effect is a monster huge SBQL query referring to the M0 version of the relational model available at the back-end.
2. The query is rewritten according to static optimization methods defined for SBQL [11] such as removing dead sub-queries, factoring out independent sub-queries, pushing expensive operators (e.g. joins) down in the syntax tree, etc. The resulting query is SBQL-optimized, but still no SQL optimization is applied.
3. According to the available information about the SQL optimizer, the back-end wrapper's mechanisms analyze the SBQL query in order to recognize patterns representing SQL-optimizable queries. Then, *exec_immediately* clauses are issued.
4. The results returned by *exec_immediately* are pushed onto the SBQL result stack as collections of structures, which are then used for regular SBQL query evaluation.

5.2 Optimization Example

The example discusses a simple two-table relational database containing information about employees `EmpR` and departments `DeptR`, “R” stands for “relational” to increase the clearness (fig. 3).

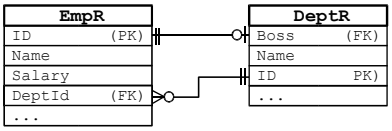

Fig. 3. The example of a relational schema

The relational schema is wrapped into an object schema shown in figure 4 according to the following view definitions. The `EmpR-DeptR` relationship is realized with `worksIn` and `boss` virtual pointers:

```
create view EmpDef {
  virtual_objects Emp {return EmpR as e;}
  virtual_objects Emp(EmpId) {return (EmpR where ID == EmpId) as e;}
  create view nameDef {
    virtual_objects name{return e.name as n;}
    on_retrieve {return n;}
  }
  create view salaryDef {
    virtual_objects salary {return e.salary as s;}
    on_retrieve {return s;}
  }
  create view worksInDef {
    virtual_pointers worksIn {return e.deptID as w;}
    on_navigate {return Dept(w) as Dept;}
  }
}

create view DeptDef {
  virtual_objects Dept {return DeptR as d;}
  virtual_objects Dept(DeptId) {return (DeptR where ID == DeptId) as d;}
  create view nameDef {
    virtual_objects name {return d.name as n;}
    on_retrieve {return n;}
  }
  create view bossDef {
    virtual_pointers boss {return e.bossID as b;}
    on_navigate {return Emp(b) as Emp;}
  }
}
```

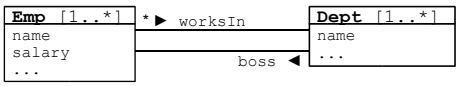

Fig. 4. Object schema used in the optimization example (wrapper's front-end)

Consider a query appearing at the front-end (visible as a business database schema) that aims to *retrieve names of the employees working in the "Retail" department with salary the same as the employee named Doe's*. The query can be formulated as follows (we assume that there is only one employee with that name in the store):

```
((Emp where worksIn.Dept.name == "Retail") where
   salary == ((Emp where name == "Doe").salary)).name;
```

The information about the local schema (the relational model) available to the wrapper that can be used during query optimization is that the `name` column is uniquely indexed and there is a primary-foreign key integrity between `DeptId` column (`EmpR` table) and `ID` column (`DeptR` table).

The optimization procedure is performed in the following steps:

1. Introduce implicit `deref` (dereference) functions
```
((Emp where worksIn.Dept.deref(name) == "Retail") where deref(salary)
== (Emp where deref(name) == "Doe").deref(salary)).deref(name);
```
2. Substitute `deref` with the invocation of `on_retrieve` function for virtual objects and `on_navigate` for virtual pointers
```
((Emp where worksIn.(Dept(w) as Dept).Dept.(name.n) == "Retail")
where (salary.s) == (Emp where (name.n) == "Doe").(salary.s)).
(name.n);
```
3. Substitute all view invocations with the queries from *sack* definitions
```
(((EmpR as e) where ((e.deptID as w).(((DeptR where ID == w) as d) as
Dept)).Dept.((d.name as n).n) == "Retail") where ((e.salary as s).s)
== ((EmpR as e) where ((e.name as n).n) == "Doe").((e.salary as s).
s)).((e.name as n).n);
```
4. Remove auxiliary names `s` and `n`
```
(((EmpR as e) where ((e.deptID as w).(((DeptR where ID == w) as d) as
Dept)).Dept.(d.name) == "Retail") where (e.salary) == ((EmpR as e)
where (e.name) == "Doe").(e.salary)).(e.name);
```
5. Remove auxiliary names `e` and `d`
```
((EmpR where ((deptID as w).((DeptR where ID == w) as Dept)).
Dept.name == "Retail") where salary == (EmpR  where name == "Doe").
salary).name;
```
6. Remove auxiliary names `w` and `Dept`
```
((EmpR where (DeptR where ID == deptID ).name == "Retail") where
salary == (EmpR  where name == "Doe").salary).name;
```
7. Now take common part before loop to prevent multiple evaluation of a query calculating salary value for Emp named *Doe*
```
((((EmpR where name == "Doe").salary ) group as z).(EmpR where
((DeptR where ID == deptID).name == "Retail")) where salary == z).
name;
```
8. Connect where and navigation clause into one where connected with and operator
```
((((EmpR where name == "Doe").salary ) group as z).(EmpR where (DeptR
where (ID == deptID and name == "Retail")) where salary == z).name;
```
9. Because `name` column is uniquely indexed, the sub-query `(EmpR  where name == "Doe")` can be substituted with *exec_ immediately* clause
```
(((exec_immediately("SELECT salary FROM EmpR WHERE name = 'Doe'"))
group as z).(EmpR where (DeptR where (ID == deptID and name ==
"Retail")) where salary == z).name;
```
10. Because the integrity constraint with `EmpR.DeptId` column and `DeptR.ID` column is available to the wrapper, the pattern is detected and another *exec_immediately* substitution is performed:
```
(((exec_immediately("SELECT salary FROM EmpR WHERE name = 'Doe'"))
group as z).(exec_immediately("SELECT * FROM EmpR, DeptR WHERE
EmpR.deptID = DeptR.ID AND DeptR.name = 'Retail'") where salary ==
z).name;
```

Either of the SQL queries invoked by *exec_immediately* clause is executed in the local relational resource and pends native optimization procedures (with application of indices and fast join, respectively).

6. Conclusions

We have presented the approach to wrapping relational databases to an object-oriented business model. The approach assumes the stack-based approach, its query

language SBQL, updatable views and the query modification technique. As shown in the example, a front-end SBQL query can be modified and optimized with application of SBA rules and updatable views within the wrapper and then the native relational optimizers for SQL language can be employed. The described wrapper architecture enables building generic solutions allowing presentation of data stored in various relational resources as object-oriented models visible at the top level of the grid and accessing the data with object query language.

The described optimization process assumes correct relational-to-object model transformation (with no loss of database logic) and accessibility of the relational model optimization information such as indices and/or primary-foreign key relations.

The method is currently being implemented as a part of our new project ODRA devoted to Web and grid applications.

References

1. Bergamaschi, S., Garuti, A., Sartori, C., Venuta, A.: Object Wrapper: An Object-Oriented Interface for Relational Databases. EUROMICRO 1997, pp.41-46
2. Foster, I., Kesselman, C., Nick, J., Tuecke, S.: The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Global Grid Forum, June 22, 2002
3. Kaczmarski, K., Habela, P., Subieta, K.: Metadata in a Data Grid Construction. Proc. of 13th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE-2004), Italy, June, 2004
4. Kozakiewicz, H., Leszczyłowski, J., Płodzień, J., Subieta, K.: Updateable Object Views. ICS PAS Reports 950, October 2002
5. Kozankiewicz, H., Stencel, K., Subieta, K.: Implementation of Federated Databases through Updateable Views. Proc. EGC 2005 - European Grid Conference, Springer LNCS, 2005, to appear
6. Kozankiewicz, H., Stencel, K., Subieta, K.: Integration of Heterogeneous Resources through Updatable Views. Workshop on Emerging Technologies for Next generation GRID (ETNGRID-2004), 13th IEEE WETICE-2004, University of Modena and Reggio Emilia, Italy, June 14-16, 2004, Proceedings published by IEEE
7. Kozankiewicz, H., Subieta, K.: SBQL Views - Prototype of Updateable Views. ADBIS (Local Proceedings) 2004
8. Matthes, F., Rudloff A., Schmidt, J.W., Subieta, K.: A Gateway from DBPL to Ingres. Proc. of Intl. Conf. on Applications of Databases, Vadstena, Sweden, Springer LNCS 819, pp.365-380, 1994
9. Moore, R., Merzky, A.: Persistent Archive Concepts. Global Grid Forum GFD-I.026. December-2003
10. Object Data Management Group: The Object Database Standard ODMG, Release 3.0. R.G.G.Cattel, D.K.Barry, Ed., Morgan Kaufmann, 2000
11. Plodzien, J.: Optimization Methods in Object Query Languages, PhD Thesis. IPIPAN, Warszawa 2000
12. Subieta, K.: Theory and Construction of Object-Oriented Query Languages. Editors of the Polish-Japanese Institute of Information Technology, 2004, 522 pages
13. Subieta, K., Plodzien, J.: Object Views and Query Modification, (in) Databases and Information Systems (eds. J. Barzdins, A. Caplinskas), Kluwer Academic Publishers, pp. 3-14, 2001
14. W3C: XQuery 1.0: An XML Query Language. W3C Working Draft 12, November 2003, http://www.w3.org/TR/xquery/