Title:

Implementation and Testing of SBQL Object-Relational Wrapper Supporting Query Optimisation

Authors:
Jacek Wiślicki, Kamil Kuliberda, Tomasz Marek Kowalski, Radosław Adamus, Kazimierz Subieta

Affiliations:
Polish-Japanese Institute of Information Technology, Warsaw, Poland
Computer Engineering Department, Technical University of Lodz, Lodz, Poland

Addresses:
Polish-Japanese Institute of Information Technology, ul. Koszykowa 86, 02-008 Warsaw, Poland
Computer Engineering Department, Technical University of Lodz, ul. Stefanowskiego 18/22, 90-924, Lodz, Poland

Emails:
jacenty@kis.p.lodz.pl, kamil@kis.p.lodz.pl, tkowals@kis.p.lodz.pl, radamus@kis.p.lodz.pl subieta@pjwstk.edu.pl

Abstract.

The paper presents design, implementation and prototype tests of an object-oriented wrapper enabling virtual integration of relational databases to an object-oriented database. The integration process is transparent - the users need not be aware that data are delivered by a relational database. The object-oriented model consists of virtual objects that can be in turn arbitrarily transformed by object-oriented updateable views, according to the virtual repository schema. Virtually transformed relational data can be combined in queries with purely object-oriented data and with data from other wrapped resources. In this implementation much attention was devoted to effective query optimisation. The described prototype employs dedicated query rewriting methods so both object-oriented and relational query optimisers can work together. Sample optimisation results and comparisons are presented.

Implementation and Testing of SBQL Object-Relational Wrapper Supporting Query Optimisation*

Jacek Wiślicki, Kamil Kuliberda, Tomasz Marek Kowalski,
Radosław Adamus, Kazimierz Subieta

Polish-Japanese Institute of Information Technology, Warsaw, Poland
Technical University of Lodz, Lodz, Poland
{jacenty, kamil, tkowals, radamus}@kis.p.lodz.pl
subieta@pjwstk.edu.pl

Abstract. The paper presents design, implementation and prototype tests of an object-oriented wrapper enabling virtual integration of relational databases to an object-oriented database. The integration process is transparent - the users need not be aware that data are delivered by a relational database. The object-oriented model consists of virtual objects that can be in turn arbitrarily transformed by object-oriented updateable views, according to the virtual repository schema. Virtually transformed relational data can be combined in queries with purely object-oriented data and with data from other wrapped resources. In this implementation much attention was devoted to effective query optimisation. The described prototype employs dedicated query rewriting methods so both object-oriented and relational query optimisers can work together. Sample optimisation results and comparisons are presented.

1 Introduction

Object-relational wrappers (or mappers, ORM) for integrating relational databases within object-oriented environments are developed for about fifteen years. A commonly recognized reason for this effort is impedance mismatch between relational databases (and SQL) and object-oriented models having specific programming constructs. The most recognized products of this development are JDO, EJB, Toplink, Hibernate and other solutions for Java and .Net, including native queries [1].

There is also a lot of doubts concerning feasibility of the goals that are pursued by ORM-s. The doubts can be summarised by two points: (1) limitations of an object-oriented model that are forced by the fact that the target data is relational; (2) performance penalty implied by ORM in case of very large relational databases. These doubts are closely related to each other. An attempt to restore an object-oriented model (e.g. originally developed in UML) from a target relational database requires non-trivial mappings which correspond to sophisticated views in databases and to the well-recognized view updating problem. This is a challenging conceptual problem that many people believe is impossible to solve for a general case. But even this problem would be solved for a less sophisticated case, the performance problem still persists. It could be impossible (or, at least, very challenging) to translate all the requests (including updates) to an object-oriented schema into SQL in such a way that SQL internal optimizers can work efficiently. The mapping that participates in this translation causes many limitations of the final SQL statements, up to the case when these statements have the trivial form *select * from R* and the results of the statements are processed sequentially by cursors or

* This work is supported by European Commission under the 6th FP project e-Gov Bus, IST-4-026727-ST

iterators. Such SQL statements give no chance to SQL optimizers. Processing large relational databases without query optimization is unacceptable for majority of business applications.

In effect, mapping between a relational schema and an object-oriented schema, as well as mapping between object-oriented queries and SQL, must be straightforward or trivial for very large relational databases. Actually, the mapping is usually isomorphic, up to secondary syntactic differences. In this way an object-oriented model is still a "slave" [2] of the relational model. For many reasons, including seamless transition from an object-oriented analysis and design model (e.g. in UML) into object-oriented implementation model (e.g. in Java) such a "slavery" is contradictory to fundamental tenets and promises of the object-orientedness as a software engineering paradigm.

Independently of the wishes of some object-oriented fans, relational databases are here and will be the most popular for many years from now. Hence the fundamental question how to obtain full freedom concerning the mapping between relational and object-object oriented schemas without compromising performance?

In our investigations within the European project eGov Bus [3] we have struggled with the problem. Our task in this project concerned creating a generic software for making *virtual repositories* (VR-s). A VR has to integrate virtually existing (legacy) data and service resources, including relational databases, XML and RDF files, data available via Web Services, etc. An essential assumption of a VR is reducing the global conceptual data and service schema to the conceptual (perhaps minimal) form that is required by the given global applications. Hence, a VR has to achieve many forms of data organization and access transparency, including distribution, heterogeneity, data representation, redundancy, replication and fragmentation transparencies. Ideologically, a VR reminds a CORBA bus, but with several conceptual changes that include richer (UML-like) data model with explicit nested collections, access to virtual objects via a query language rather than through low-level programming, the possibility to build programming and database abstractions (procedures, functions, classes, methods, views, etc.), flexibility concerning external schemata for client applications, and others. The detailed description of the software for making VR-s is the subject of other documents (see [4]). In this paper we focus only on the part that is devoted to mapping relational databases to an object-oriented schema. All the research and development is based on the ODRA system (Object Database for Rapid Application development) [4], the object-oriented query and programming language SBQL (Stack-Based Query Language) [5] and the new concept of virtual updatable views [4, 6, 7] that essentially extends the capabilities of SBQL. Just due to the power of our views we have started to describe the ODRA system as Virtual Repository Management System (VRMS).

What distinguishes our object-relational wrapper from many similar existing concepts and implementations? In contrast to all previous proposals concerning views (including SQL), our views are defined by programming functions having the full algorithmic power. The power concerns the mapping from a stored data into virtual objects, as well as the reverse mapping of updating operations addressing virtual objects into updating operations addressing the stored data (thus our idea reminds *instead of* trigger views of SQL, but it is based on different assumptions). This power causes that SBQL views make it possible to achieve effects that so far have not even been considered in the database domain.

Besides SBQL and virtual object-oriented views our idea is based on the query modification technique 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, rewriting, indices and fast joins. Due to JDBC currently supported RDBMS include Axion, Cloudscape, DB2, DB2/AS400, Derby, Firebird, Hypersonic, Informix, InstantDB, Interbase, MS Access, MS SQL, MySQL, Oracle, Postgres, SapDB, Sybase and Weblogic.

The idea of this wrapper is that a relational database is treated as a primitive object-oriented database, where each tuple is considered an object. Then, on such a primitive object-oriented database the developer of a VR can define virtual views that map it to the given object-oriented model assumed by the global object-oriented schema. SBQL queries invoke the object-oriented model, thus the relational database structure is fully transparent for its users. Due to the power of SBQL views, any complex mapping between a relational schema and an object-oriented canonical schema is feasible. The processing of SBQL queries addressing a virtual object base is done by the following steps:

- SBQL query is compiled and then its abstract syntax tree (AST) is produced;
- Each node in this AST that contains a view invocation is substituted by the view AST; this method is known as *query modification*;
- In the result we obtain a huge AST representing an SBQL query with no view invocations and addressing the relational database. This query address just the primitive object database that is 1:1 compatible with the relational database. It is first optimized by the SBQL engine by removing dead subqueries, factoring out independent subqueries from loops, etc.;
- The resulting syntactic tree cannot be entirely mapped to SQL, because SBQL is much more powerful than SQL and SBQL queries can refer to a local environment, unavailable for SQL. Hence, the tree is traversed in order to discover largest subtrees that are 1:1 compatible with SQL queries;
- Such subtrees are then mapped into SQL code using the JDBC interface;
- Then the tree is compiled to the SBQL bytecode and executed. The results from JDBC invocations are converted to the SBQL format and stored at SBQL stacks.

Benchmarks have shown that this algorithm behaves quite well and is able to utilize almost all native SQL optimization methods.

Due to the size limit in this paper we do not discuss all the details of this implementation, e.g. updating capabilities. Instead, real-life results are presented. In this paper we also omit the description of SBA, SBQL, ODRA and the VR software due to dozens of other sources; see [4, 5].

The concept of wrappers and mediators [8] to heterogeneous distributed data sources was introduced as a vision and an indication for developing future information systems for next years. The approach and the methodology presented in [8] had several implementations, e.g. Pegasus [9], Amos [10], Amos II [11] and DISCO [12]. The mediation technique is crucial for the described wrapper. Nevertheless, there exist many other approaches aiming to build object views over relational data, e.g., the variety of ORM/DAO implementations, XML views over relational data (e.g. XPERANTO [13], XTABLES [14]), also with data updating capabilities, some experimental applications of RDF and SPARQL (e.g. [15]).

The rest of the paper is organized as follows. Section 2 contains the description of the implemented wrapper architecture and query processing. Section 3 deals with query optimisation, which is explained by examples. It also presents optimisation results. Section 4 presents our conclusion.

2 Wrapper Architecture and Query Processing

Fig.1 shows the wrapper component within a virtual repository. An ODRA resource denotes any data resource providing an interface capable of executing SBQL queries and returning SBQL objects as their results (a local query optimisation should be performed also, if possible). The nature of such a resource is irrelevant, as only the mentioned capability is

important. In the simplest case, where a resource is an ODRA database, its interface has direct access to a data store and it is identical with an ODRA database engine. However, as the virtual repository aims to integrate existing business resources, whose models are mainly relational ones, an interface becomes much more complicated, as there is no directly available data store – SBQL result objects must be created dynamically basing on results returned from SQL relational queries evaluated directly in a local RDBMS.

Such a case (most common in real-life applications) forces introducing an additional middleware, i.e. a wrapper realized in the client-server architecture (Fig.1), which assures simplicity of implementation, maintenance, portability and distribution. A regular ODRA database can be extended with as many wrappers as needed (e.g., for relational or semistructured data stores, RDF resources, etc.) and plugged into any resource model without any lost of its primary performance.

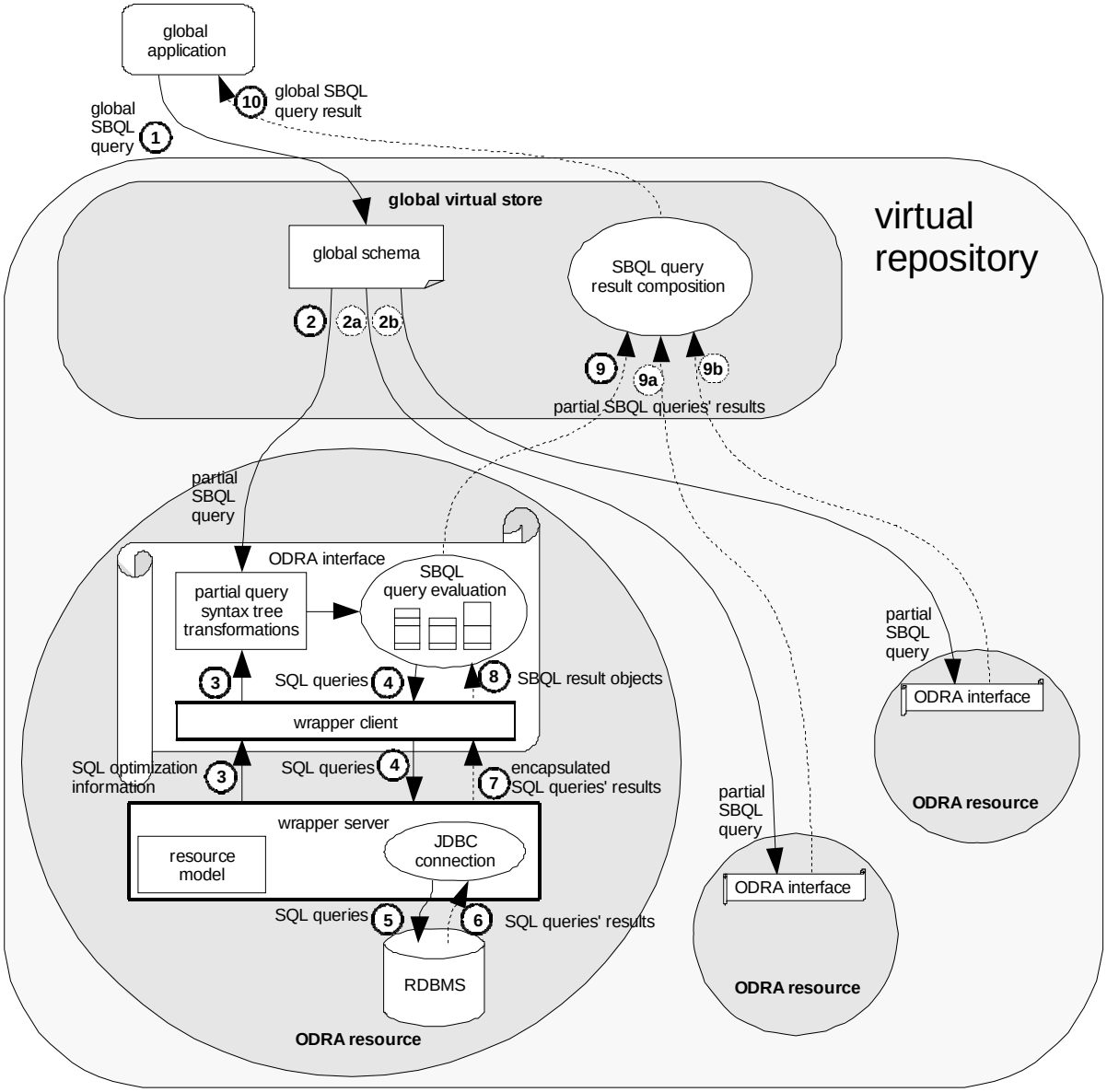

Fig. 1. Virtual repository and wrapper query processing

A query evaluation process with application of the wrapper is depicted in details in Fig.1. One of global applications sends a query (arrow 1). This query is expressed with SBQL, as it

refers to the business object oriented model available to global (top-level) users. According to the global schema and its information on data fragmentation, replication and physical location (obtained from the integration schema and from the global index), the query is decomposed into subqueries (*partial queries*) and sent to appropriate resources. This stage is realized with arrows 2, 2a and 2b. A notion "partial query" is general, as in some cases each resource-oriented query can be the same (e.g., in case of a pure horizontal data fragmentation), however in most situations partial queries are different (issues of data fragmentation and integration are discussed and the possible solutions presented in [16, 17]). Query processing corresponding to arrows 2a and 2b is out of the scope of the paper as those resources are regarded here as black boxes (*any* resources conforming with the repository requirements, including wrapped relational ones described here).

The partial query aiming at our relational resource is further processed with a resource's ODRA interface. As mentioned above, a local interface does not have its physical data store; it can only retrieve required data from its RDBMS on-the-fly. First, the interface performs query optimisation. Besides efficient SBQL optimisation rules applied at any resource's interface, here one can also transform queries so that powerful native SQL optimisers can work and amounts of data retrieved from the RDBMS are acceptably small (just matching selection criteria). Relational optimisation information (indices, cardinalities, primary-foreign key relationships, etc.) is provided by the wrapper server's resource model (arrow 3) and appropriate SBQL query syntax tree transformations are performed. These transformations are based on finding in the SBQL syntax tree patterns corresponding to SQL-optimisable queries. Appropriate tree branches (responsible for such SQL queries) are substituted with calls to *execute SQL* expression with optimisable SQL queries. Their evaluation at the resource is fast and efficient and returned results are acceptably small. However some queries may not be much optimised due to their nature excluding efficient transformation to optimal forms.

Once syntax tree transformations are finished, the interface starts a regular SBQL query evaluation. Whenever it finds an *execute SQL* procedure, its SQL query is sent to the wrapper server via the client (arrow 4, the client passed SQL queries without any modification). The server executes SQL queries as a resource client (JDBC connection), arrow 5, and their results, arrow 6, are encapsulated and sent to the client (arrow 7). Subsequently, the wrapper client creates SBQL result objects from results returned from the server (it cannot be accomplished at the resource site, which is another crucial reason for the client-server architecture) and puts them on regular SBQL stacks for further evaluation (arrow 8).

Having finished local evaluation, the interface sends its *partial result* upwards (arrow 9), where it is combined with results returned from other resources (arrows 9a and 9b) and the global query result is composed (depending on fragmentation types, redundancies and replication). This result is returned to the global application (arrow 10).

The procedure of wrapping relational schemata is not crucial for understanding the query rewriting examples provided below (it is important to virtual repository and local resource administrators), nevertheless it explains some naming conventions.

The first step consists of creating an ODRA metabase mapping one-to-one the relational schema. Over this metabase there are automatically generated basic wrapper views and virtual pointer objects created for primary-foreign key pairs. During this automatic view generation stage, every "relational" name is given a corresponding object-oriented view (names are postfixed with the *View* word, to avoid name conflicts). The resulting view-based schema (realising the SBA AS0 model) is ready for direct querying or integration to the virtual repository via arbitrarily complex integration and contributory views.

The example of the mapping procedure is illustrated on the test relational schema used for the following optimisation examples. This schema consists of three tables (*employees*, *departments* and *locations*) related by primary-foreign key constraints (an employee works in

some department that in turn is located in some town) populated with random (however consistent) data. Each table has its primary key column (*id*); there are also non-unique indices on employees' surnames and salaries, departments' names and locations' names.

The wrapping procedure is shown in the following figures. Fig.2 presents the plain relational database schema with tables and their relations shown.

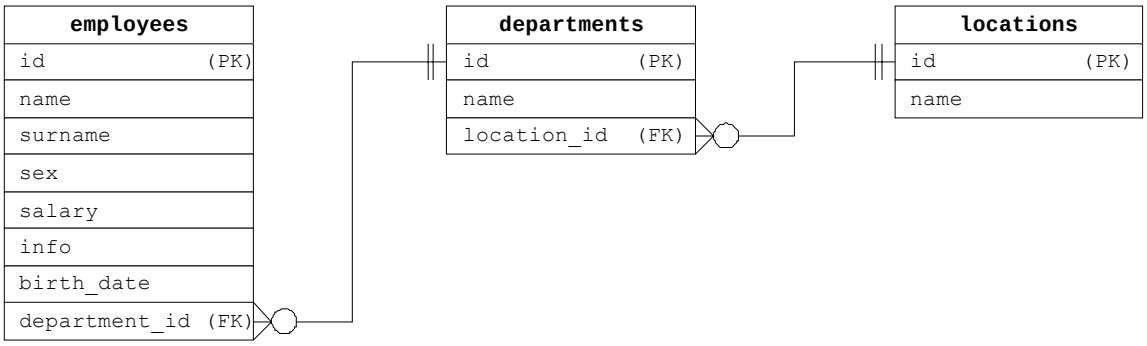

Fig. 2. Test relational schema

The result of the primitive wrapper mapping is depicted in Fig.3 – three complex metaobjects are created in ODRA metabase.

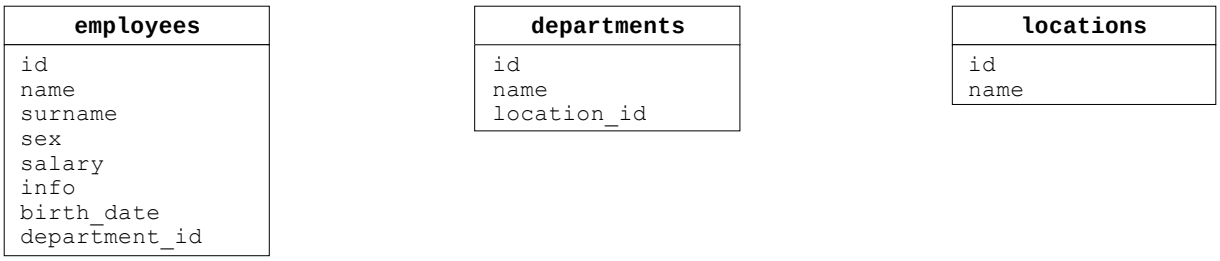

Fig.3. Object-oriented schema mapping one-to-one relational tables and columns

These metaobjects are then covered with views realising the M0 SBA model (Fig.4) – virtual pointers are created for foreign key columns to realize the wrapped schema integrity (however these pointers can be easily overridden by upper-level virtual repository views).

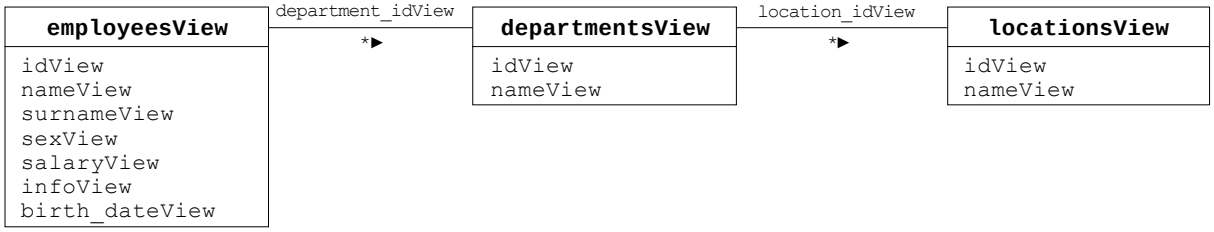

Fig. 4. Object-oriented schema covered by automatically generated wrapper views

3 Query Optimisation Methods, Examples and Results

Besides reliable and transparent integration procedures, the crucial challenge of the described solution is how to utilize a native powerful SQL optimiser in order to push query evaluation down to the wrapped resource, to minimize data throughput over network and to avoid unnecessary ODRA store space consumption. In all known RDBMSs the optimiser and its

particular structures (e.g., indices) are transparent to the SQL users. A naive implementation of a 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 optimiser has no chance to work. This problem has been solved successfully and most of query result processing is performed by a RDBMS under a wrapper. The solution is based on SBQL, virtual object-oriented views defined in SBQL, query modification methods, and dedicated optimisation mechanisms able to detect in a SBQL query syntactic tree some patterns that can be directly mapped as optimiseable SQL queries. The patterns match typical optimisation methods that are used by the SQL query optimiser, in particular, indices and fast joins.

The substantial step allowing the wrapper to access any SBQL semantics is macro-substituting view definitions for their names so that only "relational" names remain (the view rewriting and query modification procedures); the process is executed automatically as only such a form of a query can be correctly recognized and transformed by either wrapper rewriter or wrapper optimiser. Currently the wrapper is capable of performing its effective optimisation on most queries, however some SBQL expressions (e.g. procedure calls) cannot be transformed to semantically equivalent SQL structures. In such cases, in order to make query evaluation possible, relational data is retrieved to ODRA with no SQL optimisation and further processed here.

The assumed optimisation procedure relies on common rules valid for both relational and object-oriented database systems i.e. perform selections and projections as early as possible in order to minimize necessary processing (and to push the processing to the wrapped resource), IO operations (here they are substantial in the distributed system architecture) and storage. The wrapper optimiser searches the query syntax tree for expressions that can be transformed into SQL-optimisable queries (being subqueries of the SBQL query). The rewriteable expression search order is: *aggregate functions*, *joins*, *where* conditions, finally *single-table expressions* (just to determine required projected columns). This order is introduced so that the possibly largest part of the SBQL syntax tree can be replaced with the corresponding SQL query. For example, some *join* expression can contain conditions expressed with *where* expression. If *where* expressions were transformed before *join*s, the query tree would become "corrupted" and no rewritable join patter would be found afterwards. This happens because the *execute SQL* expression encapsulates the semantics of the substituted SBQL subquery/tree branch, which is enough for its correct evaluation but the reverse process would require semantic analysis of the SQL query. This is not implemented as it is currently unnecessary.

The examples presented below are based on the test schema introduced previously with the wrapping procedure description. A data used for tests are randomly generated according to some predefined distribution; therefore the optimisation results may vary for different test databases. Tests' results become more repeatable if the population of the *employees* table (number of rows is adjustable) grows up and its data distribution is closer to the assumed pattern. Nevertheless, the optimisation is always satisfactory. The presented here optimisation results are average values for 10 subsequent measurements for populations of 10, 100 and 1000 employees. The plots are presented in the log-linear scale; both the evaluation time ratio and ODRA store occupation are given.

As stated above, the wrapper optimiser searcher for the largest syntax tree branch possible for the optimisation – the result returned from the resource is indistinguishable from other objects so the ODRA query evaluation engine processes it in its regular stack-based manner. This feature enables embedding *execute SQL* expressions in larger SBQL queries with preservation of all the original typing information. These cases are not considered in the examples due to their complexity and little impact on optimisation results.

Example 1

Consider a query aiming to *retrieve surnames and names of employees earning more than 1200*. In SBQL, correspondingly to the test schema, it can be expressed as:

```
(employeesView where salaryView > 1200).(surnameView,
nameView);
```

The query modification step (views' rewriting) transforms it into the following form, where only "relational" names exist (there can appear also some auxiliary SBQL names):

```
((employees where ((salary . deref(_VALUE)) > (real)(1200)))
. (surname ,  name))
```

The query in such form is ready for wrapper optimisation (or only simple rewriting in the worst case). The optimisation procedure results in:

```
execsql("select employees.surname, employees.name from
employees where (employees.salary > 1200)", "<result
pattern>", "wrapper")
```

Here, in a simple query, only a single *execute SQL* expression appears. It keeps the SQL query string, some *result pattern* (used for keeping the SBQL query semantics and reconstructing the final result, skipped for code simplification). The SQL query string is sent to a wrapper named *wrapper* (it is specified as the last *execute SQL* parameter as they can be various wrappers mixed in a single query), executed optimally in the wrapped resource, the result is returned to the wrapper client and corresponding ODRA objects are created. The SQL query should be optimally evaluated by the wrapped resource engine as direct selection conditions and projected columns are given.

Below we present a query with no optimisation but simple rewriting only. Such a case holds if a SBQL query invokes some constructs having no counterparts in SQL.

```
((execsql("select employees.info, employees.department id,
employees.surname, employees.salary, employees.id,
employees.sex, employees.name, employees.birth date from
employees", "<result pattern>", "wrapper") where ((salary .
deref(_VALUE)) > (real)(1200))) . (surname ,  name))
```

The generated string of an SQL query retrieves all the records form the *employees* table and actual selection and projection are performed by ODRA.

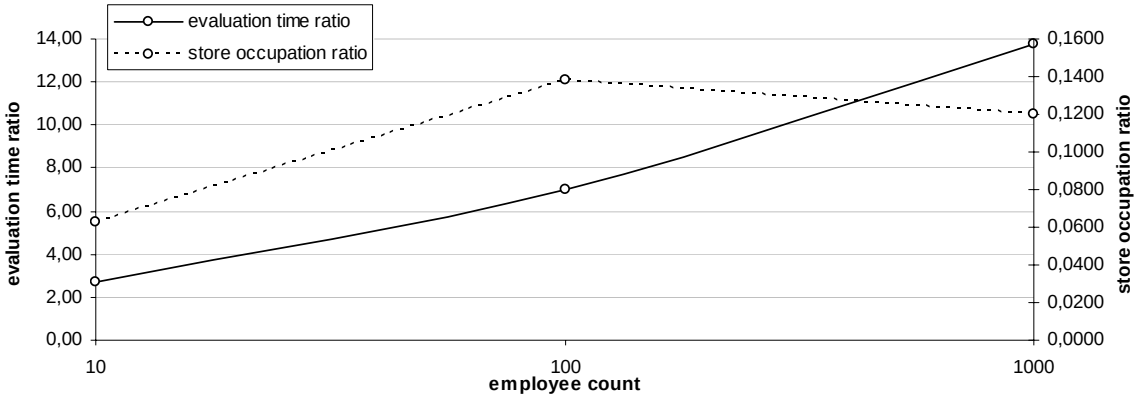

Fig. 5. Optimisation results for example 1

Example 2

Now let's introduce a more complex query combining more relational tables. It aims to *retrieve surnames of employees and cities their departments are located in* (join performed by means of a virtual pointer):

```
(employeesView as e join e.department_idView.departmentsView
as d join d.location_idView.locationsView as
l).(e.surnameView, l.nameView);
```

After views' optimisation the query is:

```
((((employees) as e join ((((e . (department_id) as
_employees_department_id) . ((departments where ((id .
deref(_VALUE)) = deref((_employees_department_id .
_VALUE))))) as departmentsView) . departmentsView)) as d)
join ((((d . (location_id) as _departments_location_id) .
((locations where ((id . deref(_VALUE)) =
deref((_departments_location_id . _VALUE))))) as
locationsView) . locationsView)) as l) . ((e . surname) ,  (l
. name)))
```

And after the wrapper optimisation:

```
execsql("select employees.surname, locations.name from
employees, locations, departments where ((departments.id =
employees.department id) AND (locations.id =
departments.location id))", "<result pattern>", "wrapper")
```

The created SQL query can be optimised and efficiently executed in the wrapped relational database since it contains join conditions expressed with primary-foreign key constraints.

Here is a non-optimised form of the query:

```
((((execsql("select employees.info, employees.department id,
employees.surname, employees.salary, employees.id,
employees.sex, employees.name, employees.birth date from
employees", "<result pattern>", "wrapper")) as e join (((e .
(department_id) as _employees_department_id) .
(execsql("select departments.name, departments.location id,
departments.id from departments", "<result pattern>",
"wrapper") where ((id . deref(_VALUE)) =
deref((_employees_department_id . _VALUE)))))) as d) join
(((d . (location_id) as _departments_location_id) .
(execsql("select locations.id, locations.name from
locations", "<result pattern>", "wrapper") where ((id .
deref(_VALUE)) = deref((_departments_location_id .
_VALUE)))))) as l) . ((e . surname) ,  (l . name)))
```

It retrieves all data from the joined tables (although no projection is required for the *departments* table as it is used only for the join operation). Join calculation and the final result projection are performed by ODRA, again.

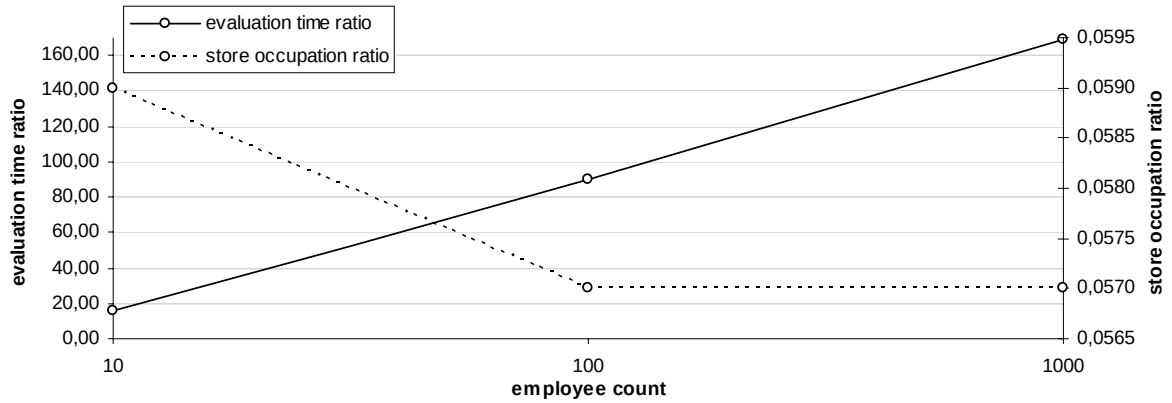

Fig. 6. Optimisation results for example 2

Example 3

The last presented query example aims to *retrieve the sum of salaries of employees named Kowalski working in Łódź city* (direct navigation through virtual pointers):

```
sum((employeesView where surnameView = "Kowalski" and
department_idView.departmentsView.location_idView.locationsVi
ew.nameView = "Łódź").salaryView);
```

After the view rewriting:

```
sum((((employees where (((surname . deref(_VALUE)) =
"Kowalski") and (((((((((department_id) as
_employees_department_id . ((departments where ((id .
deref(_VALUE)) = deref((_employees_department_id .
_VALUE)))))) as departmentsView) . departmentsView) .
(location_id) as _departments_location_id) . ((locations
where ((id . deref(_VALUE)) = deref((_departments_location_id
. _VALUE))))) as locationsView) . locationsView) . name) .
deref(_VALUE)) = "Łódź"))) . salary) . deref(_VALUE)))
```

And after the wrapper optimisation:

```
execsql("select sum(employees.salary) from employees,
locations, departments where ((employees.surname =
'Kowalski') AND ((locations.name = 'Łódź') AND
((departments.id = employees.department id) AND (locations.id
= departments.location_id))))", "<result pattern>",
"wrapper")
```

Only a single SQL query is executed and selection, projection and sum calculation are pushed to the wrapped database (ODRA receives only a number being the final result).

The worst case is when the query is non-optimised:

```
sum((((execsql("select employees.info,
employees.department id, employees.surname, employees.salary,
employees.id, employees.sex, employees.name,
employees.birth date from employees", "<result pattern>",
"wrapper") where (((surname . deref(_VALUE)) = "Kowalski")
```

```
and ((((((department_id) as _employees_department_id .
(execsql("select departments.name, departments.location id,
departments.id from departments", "<result pattern>",
"wrapper") where ((id . deref(_VALUE)) =
deref((_employees_department_id . _VALUE))))) . (location_id)
as _departments_location_id) . (execsql("select locations.id,
locations.name from locations", "<result pattern>",
"wrapper") where ((id . deref(_VALUE)) =
deref((_departments_location_id . _VALUE))))) . name) .
deref(_VALUE)) = "Łódź"))) . salary) . deref(_VALUE)))
```

Similarly as in the previous case, data from all the tables to be joined are retrieved, joins are evaluated by ODRA according to the given conditions, finally selection and projection are applied and the sum of salaries is calculated.

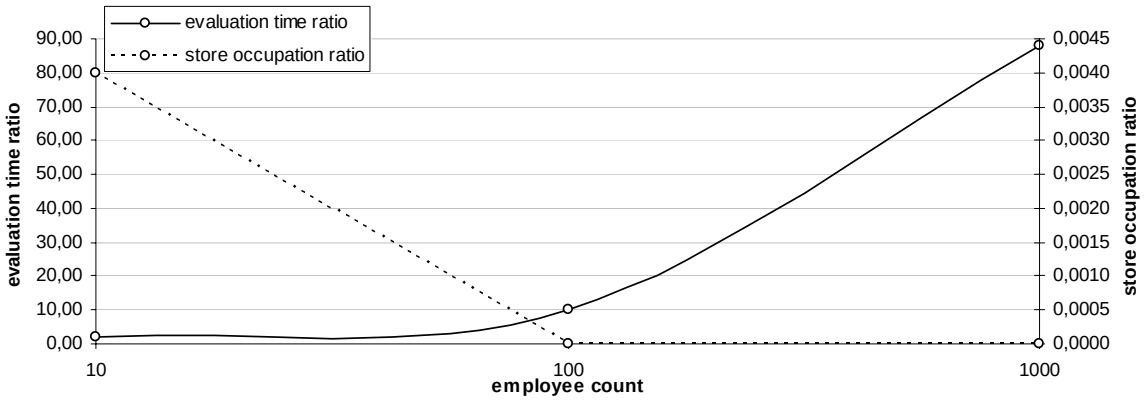

Fig. 7. Optimisation results for example 3

5 Conclusions

The implemented prototype has shown that the presented approach is feasible, functional and efficient. The main assumptions concerning transparency - no data materialisation and efficient query optimisation - are successfully achieved. Sometimes full data retrieval is required if some query cannot be optimised. Nevertheless even in such case it is evaluated correctly. However even such situations are not hopeless, as before generating SQL queries very efficient SBQL optimisation can be applied, e.g., factoring out independent subqueries [18] or additional indices. The presented architecture and methods are very flexible for wrapping not only relational databases. Currently we also work on virtual integrating other, non-relational resources, in particular, oriented towards RDF [19].

The paper size limit does not allow the description of data updating procedures, which are also available in the wrapper. In this paper we also do not discuss query rewriting procedures for multi-wrapper (i.e. referring to more than one wrapper) and mixed queries (i.e. combining wrapped objects and pure ODRA objects).

References

1. 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, 2005
2. Neward T.: The Vietnam of Computer Science, http://www.odbms.org/download/031.01%20Neward%20The%20Vietnam%20of%20Computer%20Science%20June%202006.PDF, 2006
3. eGov-Bus Project, http://www.egov-bus.org/web/guest/home, 2008
4. Virtual Repository Management System ODRA, http://www.sbql.pl/various/, 2008
5. Subieta K.: Stack-Based Architecture and Stack-Based Query Language, http://www.sbql.pl/, 2008
6. Kozankiewicz H., Leszczyłowski J., Subieta K.: Implementing Mediators through Virtual Updateable Views, Engineering Federated Information Systems, Proceedings of the 5th Workshop EFIS 2003, July 17-18 2003, Coventry, UK, pp.52-62
7. Kozankiewicz H., Stencel K., Subieta K.: Integration of Heterogeneous Resources through Updatable Views. Workshop on Emerging Technologies for Next Generation GRID (ETNGRID-2004), June 2004, Proc. published by IEEE, pp.309-314
8. Wiederhold G.: Mediators in the Architecture of Future Information Systems. IEEE Computer, 25(3), 1992, pp. 38-49
9. Ahmed R., Albert J., Du W., Kent W., Litwin W., Shan M-C.: An overview of Pegasus. In: Proceedings of the Workshop on Interoperability in Multidatabase Systems, RIDE-IMS'93, Vienna, Austria, 1993
10. Fahl G., Risch T.: Query processing over object views of relational data, The VLDB Journal (1997) 6: 261–281
11. Risch T., Josifovski, V., Katchaounov, T.: Functional data integration in a distributed mediator system. In Functional Approach to Computing with Data, P.Gray, L.Kerschberg, P.King, and A.Poulovassilis, Eds. Springer, 2003
12. Tomasic A., Raschid L., Valduriez P.: Scaling Access to Heterogeneous Data Sources with DISCO. IEEE Transactions on Knowledge and Data Engineering, Volume 10, pp 808-823, 1998
13. Carey M., Kiernan J., Shanmugasundaram J., Shekita E., Subramanian S.: XPERANTO: A Middleware for Publishing Object-Relational Data as XML Documents
14. Funderburk J. E., Kiernan G., Shanmugasundaram J., Shekita E., Wei C.: XTABLES: Bridging relational technology and XML, IBM Systems Journal. 41, No. 4, 2002
15. Perez de Laborda C., Conrad S.: Bringing Relational Data into the SemanticWeb using SPARQL and Relational.OWL, Data Engineering Workshops, 2006. Proceedings. 2006, pp. 55-55
16. Kuliberda K, Adamus R., Wiślicki J., Kaczmarski K., Kowalski T., Subieta K.: Autonomous Layer for Data Integration in a Virtual Repository, On the Move to Meaningful Internet Systems 2006, 3rd International Conference on Grid computing, high-performAnce and Distributed Applications (GADA'06), Springer 2006 LNCS 4276, Montpellier, France, 2006, pp. 1290-1304
17. Kuliberda K., Adamus R., Wiślicki J., Kaczmarski K., Kowalski T., Subieta K.: A Generic Proposal for a Transparent Integration of Distributed Data by an Autonomous Layer in a Virtual Repository, Multiagent and Grid Systems Journal (MAGS), no 4, vol 3, 2007 (to appear)
18. Płodzień J., Kraken A.: Object Query Optimization through Detecting Independent Subqueries. Information Systems, Pergamon Press, 2000
19. Petrini J., Risch T.: SWARD: Semantic Web Abridged Relational Databases, http://user.it.uu.se/~udbl/sward/SWARD.pdf