Object-Oriented Wrapper for Semistructured Data in a Data Grid Architecture[1]

Kamil Kuliberda, Jacek Wislicki, Radoslaw Adamus
Computer Engineering Department, Technical University of Lodz, Lodz, Poland
{kkulibe,jacenty,radamus}@kis.p.lodz.pl

Kazimierz Subieta
Computer Engineering Department, Technical University of Lodz, Lodz, Poland
Institute of Computer Science PAS, Warsaw, Poland
Polish-Japanese Institute of IT, Warsaw, Poland
subieta@pjwstk.edu.pl

Abstract

The paper addresses the problem of integrating data based on a XML-like semistructured model with a data grid architecture based on an object-oriented model. This work is continuation of the previous works on object-to-relational wrappers and covers a generic integration procedure for utilizing a native Lore query mechanism for retrieving data from XML data sources. A corresponding wrapper is founded on the concepts of the stack-based approach (SBA) and updatable views. The proposed architecture supports grid's transparency and allows a grid user to operate in an object-oriented environment through SBQL, a stack-based query language. The described wrapper makes it possible to employ the native Lorel query optimization mechanisms. A query entering the front-end of the wrapper (object-oriented business model) is transformed and rewritten according to the SBQL optimization rules incorporated in the wrapper and then evaluated in the native XML resource environment through the query language Lorel. The paper discusses architectural issues of such a wrapper and presents its idea through a concise example.

[1] This work was supported in part by European Commission under 6. FP project eGov-Bus, IST-4-026727-ST

1. Introduction and Motivation

The grid technology is currently widely spread among various business data processing systems. Business-oriented grids are founded on integration of heterogeneous data and service resources designed independently, based on various data models and stored with various data storage technologies. Such applications require wrappers that map local resources into the global data and service model. Through the wrappers distributed, heterogeneous and redundant resources can be virtually integrated into a centralized, homogeneous and non-redundant whole. Our gird-oriented research aims to integrate various databases, in particular implemented on relational DBMS-s, object-oriented DBMS-s, and XML-oriented DBMS. The relational-to-object wrapper is discussed in [15]. In this paper we deal with a wrapper that maps XML-like semi-structured data into an object-oriented model.

Semistructured data follow a loose data model where data may have an unfixed schema and may be irregular or incomplete. Such data arise frequently on the Web or in applications that integrate heterogeneous resources. Important cases of semistructured data are XML or RDF files that are stored under systems such Tamino, Berkeley DB or Lore in the so-called *parsed form*. Semistructured data can be neither stored nor queried in relational or object-oriented database management systems easily and efficiently [20]. In general, it is rather difficult to process semistructured data within a traditional strongly-typed data model requiring a predefined schema. Therefore many applications involving semistructured data do not use DBMS-s, despite their strengths (ad-hoc queries, efficient access, concurrency control, crash recovery, security, etc.). There is however some tendency to store XML data in relational databases and access them through additional software layers or special query languages [14]. Examples of such systems are: STORED [8], Edge [9], XPERANTO [5] and Agora [17]. In some cases these systems use sophisticated query evaluation and optimization mechanisms and our intention is to design wrappers that allow one to employ them. This relaxes us from necessity to materialize the data on the side of our grid applications.

The problems with such wrappers can be subdivided into conceptual issues and achieving proper performance. Conceptual problems concern how to develop a universal mechanism that will be able to map foreign data, perhaps developed under different data models, into a global view. This is especially difficult when the mapping must concern not only retrieval, but also updating. After solving the conceptual problems there are still performance problems. Both problems are much easier if one would assume materialization (i.e. replication) of foreign data within the grid application. Materialization of data is proposed e.g. in [4, 7]. During the materialization one can also provide changing the data to the desired schema and format required by the grid application. However, materialization of data is usually impossible due to data transmission overhead, problems with keeping consistency of materialized copies and for security policies.

Therefore we avoid materialization by direct accessing the data source and exploiting the native data model and its programming interfaces such as query languages. For performance it is critical to utilize native query optimization mechanisms. Hence during the development of the wrapper we should map somehow our grid-oriented requests into optimizable queries that are to be efficiently performed by a local resource server.

In the proposed approach we assume an object-oriented canonical data model similar to the CORBA, ODMG [5] or UML data models. The model is supported by SBQL, a query language based the Stack-Based Approach (SBA) [25]. SBQL has the algorithmic power of programming languages thus it is powerful enough to express any required properties of a common canonical data model as well as business intentions of the grid users. The clean semantics of SBQL allows us to develop optimization methods comparable or more powerful in comparison to the methods applied in SQL and in specific optimization and cost-based techniques proposed for semistructured data repositories aiming XML.

Within SBA we have developed a novel mechanism of virtual updatable object-oriented views [11]. It allows for achieving many forms of transparency that are required by grid applications. The global virtual store delivers virtual objects (defined by means of updatable views) that are indistinguishable for the grid users from the stored objects. The views allow full updatability of the virtual objects; for details see [11, 25].

The basic assumption of the presented approach is that the local data can be described with any common data model (e.g. relational, XML, data-sheet). The paper deals with the architecture of a generic wrapper for a semistructured data model based on XML data and extended through a modified data model OEM, *Object Exchange Model* [23] implemented in database management system Lore (*Lightweight Object Repository*) [19]. We exploit Lorel [1], the Lore's query language, which supports optimization methods for queries addressing XML and OEM [19, 20, 21]. The main challenge is taking advantage of the native Lorel query optimizer in the proposed grid architecture. Although queries can be optimized on the SBQL side, some optimization features, such as indices and fast joins [20, 21, 22], can be exploited only on the Lore side. The wrapper must translate somehow SBQL queries into Lore queries in such a way that these optimization features will be fully utilized.

Currently, we are implementing (under .NET and Java) an object-oriented platform named ODRA for Web and grid applications, thus the problem of integrating different foreign resources through wrappers is critical. After previous experience we have made the following assumptions:

1. The system will be based on our own object-oriented query language SBQL, which has many advantages over OQL, XQuery, SQL-99 and other languages. In particular, SBQL has the power of programming languages and precise semantics, which is a prerequisite for developing automatic transformations of queries into semantically equivalent forms. SBQL is already implemented, including a type checker and a query rewriting optimizer,

2. The system will be equipped with a powerful mechanism of object-oriented virtual updatable views based on SBQL. Our views are much more powerful than e.g. SQL views, because they are defined through constructs having the computational power of programming languages. There are three basic applications of the views: (1) as integrators (mediators) making up a global virtual data and service store on top of distributed, heterogeneous and redundant resources; (2) as wrappers on top of particular local resources; (3) as customization and security facility on top of the global virtual store. A prototype implementation of SBQL views is ready [11].

We have experience with designing a similar wrapper from the DBPL system to Ingres and Oracle, see [18] for details. The idea was that DBPL queries are to be automatically mapped to SQL queries. On the back-end the DBPL queries (mapped onto SQL) were optimized by the SQL optimizer. Another such a wrapper was implemented by us for the European project ICONS, IST-2001-32429, devoted to advanced Web applications. This wrapper was based on a rather simplified object-oriented model. Currently we are involved in the next European project eGov-Bus (Advanced e-Government Information Service Bus), IST-4-026727-ST, devoted to a dynamically adaptable information system supporting life events experienced by the citizen or business serviced by European government organizations. The eGov-Bus prototype must integrate distributed and heterogeneous resources that are under the control of various institutions. The project assumes an advanced object-oriented model comparable to UML and ODMG models, thus requires more sophisticated wrappers to local data and service resources, in particular, based on relational databases and XML.

The rest of the paper is structured as follows. Section 2 introduces the problem of processing semistructured data in a data grid. Section 3 describes the authors' idea of the grid architecture. Section 4 focuses on the architecture of a generic wrapper for semistructured data and explains the method of utilization of the Lorel optimization in the presented solution. Section 5 concludes.

2. Discussion on Semistructured Data Processing in a Data Grid Architecture

Integration of dozens or hundreds of servers participating in a grid requires different design processes in comparison to the situation where one object-oriented application is to be connected to a database with a semistructured data model. The common (canonical) grid's database schema is a result of many negotiations and tradeoffs between business partners having incompatible (heterogeneous) data and services. This makes development of an object-semistructured wrapper much more constrained than in a non-grid case. On one hand, the wrapper should deliver the data and services according to the predefined object-oriented canonical schema. On the other hand, its back-end should work on a given semistructured data store.

The major problem with this architecture concerns how to utilize native query optimizers. As we know, the access to simple XML data is hard to optimize even through native XML query languages such as XQuery. The Lore system is designed specifically for the management of such semistructured data. The data managed by Lore may not be constrained by a schema, it may be irregular or incomplete. In general, Lore attempts to take advantage of a structure wherever it exists, but it also handles irregular data as gracefully as possible. Lore is equipped with its own data model and a query language with an optimizer designed especially for semistructured data, including a cost-based optimizer [19]. The most important for us is that Lore is fully functional and available to the public. The Lore's data model called OEM (*Object Exchange Model*) [23] is a very simple self-describing graph-based nested object model. Once an XML document is mapped onto the OEM data model it is convenient to visualize the data as a directed labelled ordered graph, where nodes represent data elements and edges represent the element-subelement relationship. Each node representing a complex data element contains a tag and an ordered list of attribute-name/atomic-value pairs; atomic data element nodes contain string values. There are two different types of edges in the graph: (1) normal subelement edges, labelled with the tag of the destination subelement; (2) crosslink edges, labelled with the attribute name that introduces the crosslink (see Figure 3). Objects in this data model have unique object identifiers (OIDs) and labels. For simplification, object instances can be referred to with their label names. Objects can be divided into atomic objects (with types such as integer, real, string, gif, java, audio, etc.) and complex objects which may have outgoing edges to subobjects. Any object that cannot be accessed by a path from some name is considered to be deleted. In an OEM database there is no notion of a fixed schema. All the schematic information is included in the labels, which may change dynamically. Thus, an OEM database is self-describing, and there is no regularity imposed on the data. The model is designed to handle incompleteness of data, as well as structure and type heterogeneity.

The content of the object can be accessed by a sequence of dot-separated labels (query path expressions) defined in Lorel – *Lore Language*. It allows the users to easily retrieve and update data stored in original XML documents. Lorel is an extension of OQL [3, 6] with certain modifications and extensions that are useful when querying semistructured data. It introduces extensive type coercion and powerful path expressions for traversing semistructured data, and extensive automatic coercion for handling heterogeneous and/or typeless data without generating errors, like in SQL and OQL. Most of Lorel is functional within the Lore system, including some subqueries, aggregation and arithmetic operations, constructed results, a declarative update language, and view facilities. Lorel is described in details in [1].

Semistructured data introduces some further problems concerning techniques of indexing. In relational and object-oriented database systems indices are created over a set of collection of specified type attributes defined in advance, while Lore provides labelled directed graph where the data is essentially arbitrarily allocated. In this case it is difficult to isolate an attribute of a collection to index, and the type of an object is not known in advance. Lore performs automatic type coercion when comparing objects of different types. This is an essential feature when dealing with semistructured data. Indexing atomic values in Lore graph-based data model allows the query engine to quickly locate specific leaf objects. Inconvenience is related to long time query evaluations evoked by exploring the data via complete path of labelled traversals through the graph. A solution to this problem was creating additional indices that efficiently locate edges and paths through the data. To speed up query processing in a Lore database there are elaborated and built four different types of index structures. The two called Vindex and Tindex identify objects that have specific values and the other two called Lindex and Pindex are used to efficiently traverse the database graph. Vindex (value index), localizes atomic objects with certain values. Tindex (text index), localizes string atomic values containing specific words or groups of words. They can be built selectively over objects with certain incoming labels. The OEM does not support parent pointers, therefore Lindex (link index), localizes parents of a specific object, while Pindex (path index) provides fast access to all objects reachable via a given labelled path [19, 20, 21].

The Lore DBMS is also equipped with a cost-based query optimizer. General approach to query optimization is typical like other DBMSs, including cost estimation for concurrent evaluation plans [22] and a formal basis for deriving optimization rewriting rules such as pushing selections down. Additionally, path expression optimization exploits the mechanisms of query pruning and query rewriting using state extents.

In all known DBMSs, the optimizer and its particular structures (e.g. indices) are transparent to the query language users. A naive implementation of a wrapper causes that it generates primitive queries in given query language such as *select * from R*, and then processes the results of such queries by DBMS QL cursors. Hence, the query optimizer has no chance to work. Our experience has shown that direct translation of object-oriented queries into Lorel is infeasible for a sufficiently general case. We propose to transform some parts of object-oriented queries in SBQL into accurate optimizable query at the Lorel side.

During querying over a wrapper, the mapping between a semistructured 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.

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 deal with views. 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 implemented artifact. In our opinion, the Stack-Based Approach and its query language SBQL offer the first and universal solution to the problem of updatable 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 semistructured databases. The described architecture assumes that a semistructured database will be seen as a simple object-oriented database, where each labelled object between edges from OEM's labelled directed graph 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.

3. Architecture of a Data Grid

In this section we briefly sketch the most important elements of the proposed grid architecture and situate them inside the wrapper module. The proposed grid architecture is clearly shown in Figure 1. Our solution provides an access simplification to the distributed, heterogeneous and redundant data, constituting an interface to the distributed data residing in any local resource provider participating in a grid. The goals of the approach are to design a platform where all clients and providers are able to access multiple distributed resources without any complications concerning data maintenance and to build a global schema for the accessible data and services. The main difficulty of the described concept is that neither data nor services can be copied, replicated and maintained on the global applications side (in the global schema), as they are supplied, stored, processed and maintained on their autonomous sites [12, 13].

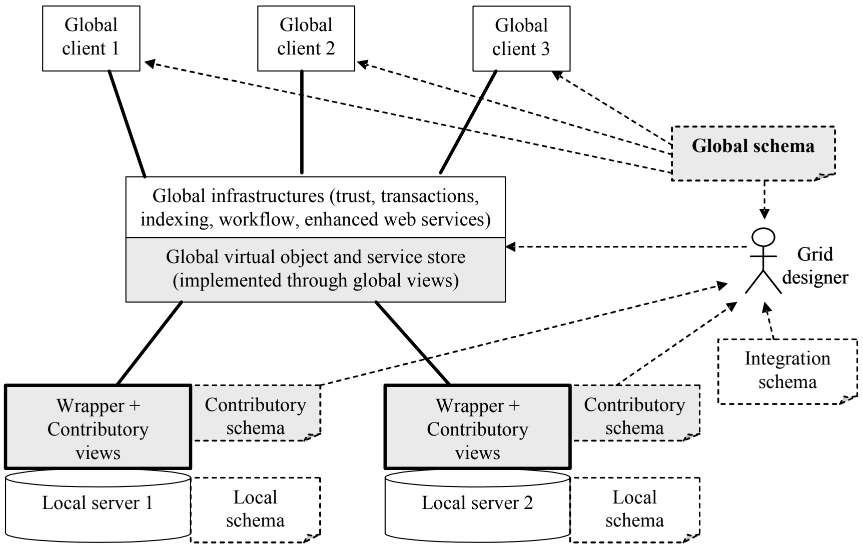

Global client 1
Global client 2
Global client 3
Global schema
Global infrastructures (trust, transactions, indexing, workflow, enhanced web services)
Global virtual object and service store (implemented through global views)
Grid designer
Integration schema
Wrapper + Contributory views
Contributory schema
Wrapper + Contributory views
Contributory schema
Local server 1
Local schema
Local server 2
Local schema

Figure 1. The data grid architecture.

Following Figure 1, the elements filled in grey realize the main aspects of contribution and integration processes of the data mapping from the wrappers [12]. The central part of a grid is a *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, as well as to enable accessing the grid according to the trust infrastructure (including security, privacy, licensing and non-repudiation issues). It also presents business objects and services according to the *global schema.* A global schema has to be defined and agreed upon the organization that creates a grid. This is a principal mechanism enveloping all the local resources into one global data structure. Physically, it is a composition of the contribution schemata which can participate in the grid. The *global schema* is responsible for managing grid contents through access permissions, discovering resources, controlling location of resources, indexing whole grid attributes. The global schema is also used by programmers to create global client applications.

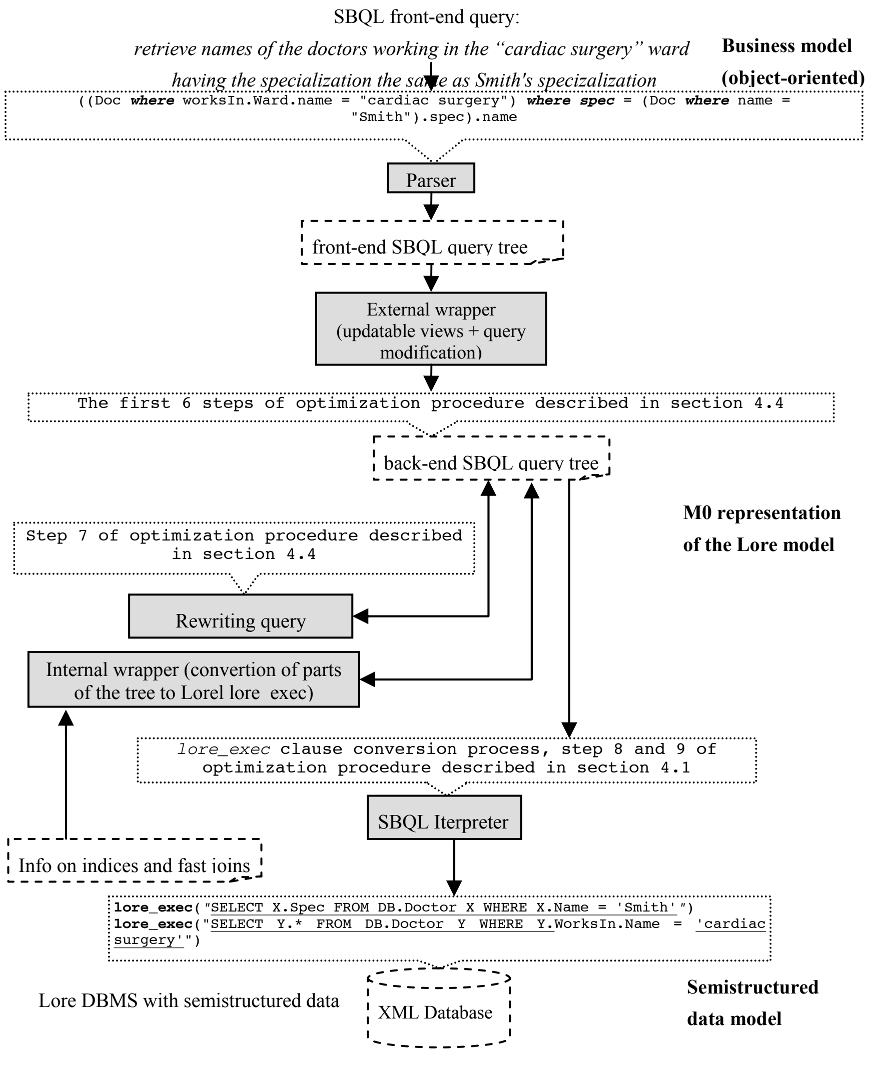
Administrators of *local servers* must define *contributory schemata* and corresponding *contributory views* [10, 12] for integrating services and objects physically available from local servers and mapping local data to the global schema. A contribution schema is created by the *grid designer* and represents the main data formalization rules for any local resource. Basing on this, local resource providers create their own contribution schemata adapted to a unique data structure present at their local sites. A *contribution view* is the query language definition of mapping the local schema to contribution schema. A well defined contribution view can become a part of the *global view*. The mapping process consists of enclosing particular contribution schemas residing in local sites into the global schema, created earlier by local resource providers [10]. The *integration schemata* contain additional information about dependencies between local servers (replications, redundancies, etc.) [10, 12], showing a method for integration of fragmented data into the grid, e.g. how to merge fragmented collections of object data structures where some parts of them are placed in separated local servers. A grid designer must be aware of the structure fragmentation, which knowledge is unnecessary for a local site administrator.

The crucial element of the architecture is a *wrapper* which enables importing and exporting data between different data models, e.g. our object-oriented grid solution at one side and a Lore OEM data model on the other side. The implementation challenge is a method of combining and enabling free bidirectional processing of contents of local resource providers participating in the global virtual store as parts of the global schema. The presented architecture of a data grid is fully scalable, as growing and reducing the grid contents is dependent on the state of global schema and views.

4. Wrapper

This section presents the architecture of the generic wrapper to the semistructured data stored in XML documents and querying via Lorel. Figure 2 presents the architecture of the wrapper. Externally the data are designed according to the object-oriented model and the business intention of the *contributory schema*. This part constitutes the *front-end* of the wrapper and relies on SBQL. Internally the structures are presented in the SBA M0 model [25]. This part constitutes the *back-end* of the wrapper and is also relies on SBQL.

The mapping between front-end and back-end is defined through updatable object views. They role is to map back-end into front-end for querying virtual objects and front-end onto back-end for updating them.


SBQL front-end query:
retrieve names of the doctors working in the "cardiac surgery" ward having the specialization the same as Smith's specizalization
Business model (object-oriented)
Parser
front-end SBQL query tree
External wrapper (updatable views + query modification)
The first 6 steps of optimization procedure described in section 4.4
back-end SBQL query tree
M0 representation of the Lore model
Step 7 of optimization procedure described in section 4.4
Rewriting query
Internal wrapper (convertion of parts of the tree to Lorel lore exec)
lore_exec clause conversion process, step 8 and 9 of optimization procedure described in section 4.1
SBQL Iterpreter
Info on indices and fast joins
Lore DBMS with semistructured data
XML Database
Semistructured data model

```
((Doc where worksIn.Ward.name = "cardiac surgery") where spec = (Doc where name =
"Smith").spec).name
```

```
lore_exec("SELECT X.Spec FROM DB.Doctor X WHERE X.Name = 'Smith'")
lore_exec("SELECT Y.* FROM DB.Doctor Y WHERE Y.WorksIn.Name = 'cardiac
surgery'")
```

Figure 2. The architecture of a generic wrapper for semistructured database.

The following optimization methods hold for the most often select-type queries, possibly there might be need to adjust them in case of updating or deleting queries, especially at the query modification stage. The object accessing on both sides of wrapper (front-end and back-end) is similar. At the front-end there is an object data model supported by SBQL. Each object in a database can be distinguished by a unique object identifier (*OID*).

At the back-end of the wrapper there is a semistructured data model supported by Lorel. The database objects can be identified by a pair: an object identifier (OID) and a label – the name of object collection. When Lorel needs to access a simple object, it must give both parameters <OID, label>. In this case either-side mapping of various sequences (including references, pointers) of objects is rather unsophisticated and we can call optimization methods without any limitations on both sides of the wrapper.

There is an assumption that for better performance of object mapping the wrapper designer should include in its structure information about available Lorel optimization mechanisms such as indices, cost-base optimizations, which can be introduced manually if not available automatically from the Lore catalogs.

The wrapper module should reach a semistructured data without any limitations. This obliges wrapper designers to produce a suitable connector for mapped DBMS (in this case Lore). Because Lore is available to public and their creators has issued the open API for programmers [16, 19] it is possible to create an application module which can freely and directly manage a data through Lorel. We have supplied the wrapper with *lore_exec* operation which communicates with the API application – it can send queries to Lore DBMS (for full processing including utilization of native optimizations) and receive (back to the API application) complete data collections as responses (including data properties such as OIDs and labels) within Lore DBMS.

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

- Query modification procedure [11, 24] applies to seeds defined by single queries and results in applying `on_retrieve` functions (`on_navigate` in case of pointers), i.e. all front-end query elements referring views are substituted with appropriate macros from views' definitions. The final effect is a huge SBQL query referring to the M0 model [25] available at the back-end.
- The modified query is rewritten according to static optimization methods defined for SBQL [24] such as removing dead sub-queries, method of independent queries, factoring/pushing out, etc. The resulting query is SBQL-optimized, but still no Lorel optimization is applied.
- The SBQL-optimized query can be now transformed to a form that native Lorel optimization is applicable. According to the available information about Lorel optimizer, the back-end wrapper's mechanisms analyze the SBQL query in order to recognize patterns representing Lorel-optimizable queries. For instance, if for the SBQL query of the form Y **where** X == v there is a Lorel index on Y objects and their X subobjects, it is substituted (in the syntax tree) with *lore_exec* clause invoking the appropriate Lorel query:

lore_exec(“**SELECT** z.* **FROM** DB.Y z **WHERE** z.X = v”)

Any Lorel query invoked from *lore_exec* clause is assumed to be optimized efficiently and evaluated in the native Lore DBMS environment. Its result is pushed onto the SBQL query result stack in a form of Lorel tuples stored as complex binders and used for regular SBQL query evaluation and also for update and delete-type queries.

4.1. Optimization Example

As an optimization example consider a simple structure of labelled directed graph objects stored in the Lore. The model contains information about doctors `DB.Doctor` and hospital’s wards `DB.Ward`, “DB” stands for the stub of semistructured data model depicted through labelled directed graph, see Figure 3.

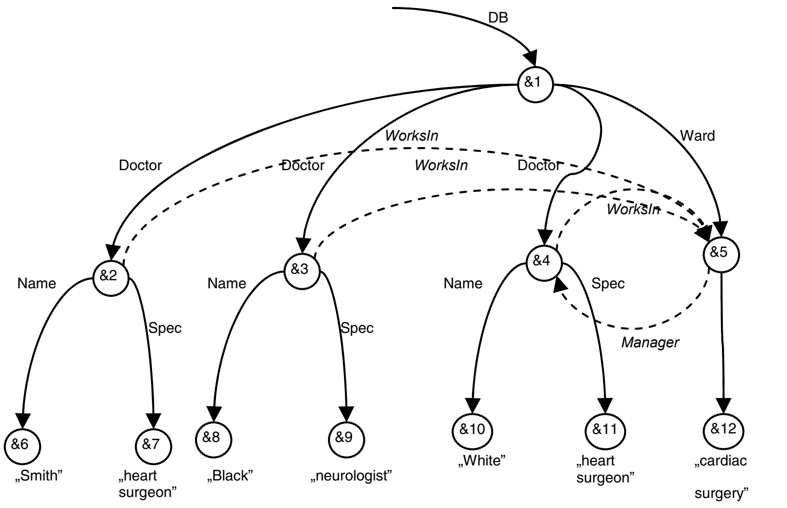


Figure 3. The example of a semistructured data model in labelled directed graph.

The semistructured schema is wrapped into an object schema shown in Figure 4 according to the following view definitions. The `DB.Doctor-DB.Ward` relationship is realized with `worksIn` and `manager` virtual pointers:

```
create view DocDef {
   virtual_objects Doc {return DB.Doctor as d;}
   create view nameDef {
      virtual_objects name{return d.Name as n;}
      on_retrieve {return n;}
   }
   create view specDef {
      virtual_objects spec {return d.Spec as s;}
      on_retrieve {return s;}
   }
   create view worksInDef {
      virtual_pointers worksIn {return d.WorksIn as wi;}
      on_navigate {return wi as Ward;}
   }}
```

```
create view WardDef {
   virtual_objects Ward {return DB.Ward as w;}
   create view nameDef {
      virtual_objects name {return w.Name as n;}
      on_retrieve {return n;}
   }
   create view managerDef {
      virtual_pointers manager {return w.Manager as b;}
      on_navigate {return b as Doc;}
    }}
```

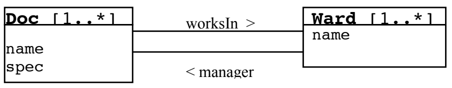


Figure 4. Object schema used in the optimization example (wrapper's front-end).

Consider a query appearing at the front-end (visible as a business database schema) that aims to retrieve names of the doctors working in the "cardiac surgery" ward having the specialization the same as Smith's specialization. The query can be formulated as follows (we assume that there is only one doctor with that name in the store):

```
((Doc where worksIn.Ward.name = "cardiac surgery") where
    spec = (Doc where name = "Smith").spec).name;
```

The information about the local schema (semistructured model) available to the wrapper that can be used during the query optimization is that `Name` objects are uniquely indexed by specific Lore indexes.

The transformation and optimization procedure is performed in the following steps:

1. First we introduce the implicit `deref` (dereference) function:
   ```
   ((Doc where worksIn.Ward.deref(name) == "cardiac surgery") where
   deref(spec) == (Doc where deref(name) ==
   "Smith").deref(spec)).deref(name);
   ```
2. Next the wrapper substitutes `deref` with the invocation of `on_retrieve` function for virtual objects and `on_navigate` for virtual pointers:
   ```
   ((Doc where worksIn.(wi as Ward).Ward.(name.n) == "cardiac
   surgery") where (spec.s) == (Doc where (name.n) ==
   "Smith").(spec.s)).(name.n);
   ```
3. The wrapper substitutes all view invocations with the queries from *virtual objects* definitions:
   ```
   (((DB.Doctor as d) where ((d.WorksIn as wi).(wi as
   Ward)).Ward.((w.Name as n).n) == "cardiac surgery") where
   ((d.Spec as s).s) == ((DB.Doctor as d) where ((d.Name as n).n)
   == "Smith").((d.Spec as s).s)).((d.Name as n).n);
   ```
4. Then it removes auxiliary names `s` and `n`:
   ```
   (((DB.Doctor as d) where ((d.WorksIn as wi).(wi as
   Ward)).Ward.(w.Name) == "cardiac surgery") where (d.Spec) ==
   ((DB.Doctor as d) where (d.Name) == "Smith").(d.Spec)).(d.Name);
   ```
5. Also it removes auxiliary names `d` and `w`:
   ```
   ((DB.Doctor where ((WorksIn as wi).(wi as Ward)).Ward.Name ==
   "cardiac surgery") where Spec == (DB.Doctor where (Name ==
   "Smith").Spec).Name;
   ```
6. Next it removes auxiliary names `wi` and `Ward`:
   ```
   ((DB.Doctor where WorksIn.Name == "cardiac surgery") where Spec
   == (DB.Doctor where (Name == "Smith").Spec).Name;
   ```

7. Now the common part is taken before the loop to prevent multiple evaluation of a query calculating specialization value for the doctor named *Smith*:
```
((((DB.Doctor where Name == "Smith").Spec) group as z).
(DB.Doctor where WorksIn.Name == "cardiac surgery") where Spec
== z).Name;
```
8. Because `Name` objects are uniquely indexed (in path `DB.Doctor`), the sub-query (*`DB.Doctor where Name == "Smith"`*) can be substituted with the *lore_exec* clause:
```
(((lore_exec("SELECT X.Spec FROM DB.Doctor X WHERE X.Name =
'Smith'")) group as z).(DB.Doctor where WorksIn.Name == "cardiac
surgery") where Spec == z).Name;
```
9. The same situation can be performed for the sub-query (*`DB.Doctor where WorksIn.Name == "cardiac surgery"`*). The wrapper produces the following *lore_exec* substitution:
```
(((lore_exec("SELECT X.Spec FROM DB.Doctor X WHERE X.Name =
'Smith'")) group as z).(lore_exec("SELECT Y.* FROM DB.Doctor Y
WHERE Y.WorksIn.Name = 'cardiac surgery'") where Spec ==
z).Name;
```

The presented above Lorel queries invoked by *lore_exec* clause are executed in the local data resource.

5. Conclusions

We have shown that the presented approach to wrapping databases based on semistructured data to object-oriented business model with application of the stack-based approach and updatable views is conceptually feasible, clear and implementable. As it is shown in the example, a front-end SBQL query can be modified and optimized by application of appropriate SBA rules and methods within the wrapper (updatable views) and then by the native optimizers for an appropriate resource query language. The described wrapper architecture enables one to build generic solutions allowing virtual representation of data stored in various resources as objects in an object-oriented model.

The described optimization process assumes correct semistructured-to-object model transformation (with no loss of database logic) and accessibility of the semistructured model optimization information such as indices and/or cost-based optimizations. The native resource access optimizer can be fully utilized by the proposed method.

6. References

1. S.Abiteboul, D.Quass, J.McHugh, J.Widom, and J.Wiener. The Lorel Query Language for Semistructured Data. Intl. Journal on Digital Libraries, 1(1):68-88, April 1997.
2. S.Abiteboul, R.Goldman, J.McHugh, V.Vassalos, and Y.Zhuge. Views for semistructured data. Proc. of the Workshop on Management of Semistructured Data, pages 83-90, Tucson, Arizona, May 1997.

3. F.Bancilhon, C.Delobel, and P.Kanellakis, eds. Building an Object-Oriented Database System: The Story of O2. Morgan Kaufmann, San Francisco, California, 1992.
4. C.K.Baru, A.Gupta, B.Laudascher, R.Marciano, Y.Papaconstantinou, P.Velikhov, V.Chu. XML-based information mediation with MIX, Proc. ACM SIGMOD Conf. on Management of Data, 1999.
5. M.Carey, J.Kiernan, J.Shanmugasundaram, E.Shekita, S.Subramanian: XPERANTO: A Middleware for Publishing Object-Relational Data as XML Documents, Proc. of the 26th VLDB Conf., 2000.
6. Object Data Management Group: The Object Database Standard ODMG, Release 3.0. R.G.G.Cattel, D.K.Barry, Ed., Morgan Kaufmann, 2000.
7. V.Christophides, S.Cluet, and J.Simeon. On wrapping query languages and efficient XML integration, In Proc. of ACM SIGMOD Conf. on Management of Data, 2000.
8. A. Deutsch, M Fernandez, D.Suciu. Storing semistructured data with STORED, Proc. of SIGMOD, 1999.
9. D.Florescu, D.Kossman. Storing and Querying XML Data using an RDBMS. Data Engineering Bulletin, 22(3), 1999.
10. K.Kaczmarski, P.Habela, K.Subieta. Metadata in a Data Grid Construction. Workshop on Emerging Technologies for Next generation GRID (ETNGRID-2004), 13th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE-2004), 2004.
11. H.Kozankiewicz, J.Leszczyłowski, J.Płodzień, K.Subieta. Updateable Object Views. ICS PAS Reports 950, October 2002
12. H.Kozankiewicz, K.Stencel, K.Subieta. Integration of Heterogeneous Resources through Updatable Views. Workshop on Emerging Technologies for Next generation GRID (ETNGRID-2004), 13th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE-2004), 2004.
13. H.Kozankiewicz, K.Stencel, K.Subieta. Implementation of Federated Databases through Updateable Views. Proc. 2005 European Grid Conference, Springer LNCS, 2005
14. R.Krishnamurthy, R.Kaushik, J.F.Naughton. XML-to-SQL Query Translation Literature: The State of the Art and Open Problems. Proc. of the 1st Int'l XML Database Symposium (XSym), pages 1-18, Berlin, Germany, September 2003.
15. K.Kuliberda, J.Wislicki, R.Adamus, K.Subieta. Object-Oriented Wrapper for Relational Databases in the Data Grid Architecture. OTM Workshops 2005, LNCS 3762, Springer 2005, pp. 367-376
16. Lore DBMS Web Page: http://www-db.stanford.edu/lore/
17. I.Manolescu, D.Florescu, D.Kossmann, F.Xhumari, D.Olteanu. Agora: Living with XML and Relational, Proc. of the 26th VLDB Conf., 2000.
18. F.Matthes, A.Rudloff, J.W.Schmidt, K.Subieta. A Gateway from DBPL to Ingres. Proc. of Intl. Conf. on Applications of Databases, Vadstena, Sweden, Springer LNCS 819, pp.365-380, 1994
19. J.McHugh, S.Abiteboul, R.Goldman, D.Quass, and J.Widom. Lore: A Database Management System for Semistructured Data. SIGMOD Record, 26(3):54-66, 1997
20. J.McHugh, J.Widom, S.Abiteboul, Q.Luo, and A.Rajaraman. Indexing Semistructured Data. Technical Report, January 1998.
21. J.McHugh, J.Widom. Query Optimization for XML. Proc. of 25th Intl. VLDB Conf., Edinburgh, Scotland, September 1999.
22. J.McHugh, J.Widom. Query Optimization for Semistructured Data. Technical Report, November 1997.

23. Y.Papakonstantinou, H.Garcia-Molina, and J.Widom. Object exchange across heterogeneous information sources. Proc. of the 11th International Conference on Data Engineering, pp. 251-260, Taipei, Taiwan, March 1995.
24. J.Plodzien. Optimization Methods In Object Query Languages, PhD Thesis. IPIPAN, Warszawa 2000.
25. K.Subieta. Theory and Construction of Object-Oriented Query Languages. Editors of the Polish-Japanese Institute of Information Technology, 2004 (in Polish).