JOURNAL OF APPLIED
COMPUTER SCIENCE
Vol. xx. No xx (200x), pp. xx-xx

Integration of relational resources in an object-oriented data grid with an example*

Jacek Wiślicki, Kamil Kuliberda, Tomasz Kowalski, Radosław Adamus

Technical University of Lodz
Computer Engineering Department
ul. Stefanowskiego 18/22, 90-924 Lodz, Poland
e-mail: {jacenty, kamil, tkowals, radamus}@kis.p.lodz.pl

Abstract. *The paper describes aspects of integrating relational resources in an object-oriented data intensive grid with extensive description of a grid architecture. The presented solution is generic and allows utilizing native query optimizers of a relational resource (indices, fast joins, etc.), which can be effectively combined with an object query language (SBQL) optimization.*

1. Introduction

A grid is a novel technology widely researched by many academic and industrial institutions. Formerly "a grid" referred only to computational networks, however, due to the rapid evolution of the Internet, Internet communities and the increase of a worldwide business information exchange there have arisen further expectations. Simultaneously, new opportunities has opened for data and service integration. Grids with we deal are conceptually similar to computational ones. Our research is devoted to a distributed parallel database content processing, where various data and service resources residing in separate locations can be virtually available through their global representation. This technology is referred to as a data-intensive grid or just a data grid.

Such a global representation (view) should abstract its users from all the technical aspects of the process of integration (location, heterogeneity,

* This work is supported by European Commission under the 6th FP project e-Gov Bus, IST-4-026727-ST and by European Social Fund and Polish State under "Mechanizm WIDDOK" programme (contract number Z/2.10/II/2.6/04/05/U/2/06)

fragmentation, replication, redundancy, etc.), which is referred to as a transparency. The effect of transparency requires enveloping (wrapping) the grid resources with dedicated programmatic structures – wrappers.

The paper describes object-to-relational wrapper concepts for distributed, heterogeneous and redundant data resources that are to be virtually integrated into a centralized, homogeneous and non-redundant whole. Our data grid concerns 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].

Integration of dozens or hundreds servers participating in a grid requires different design processes in comparison to the situation when, e.g. a single object-oriented application has to be connected to a relational database. A common (canonical) database schema is the result of many negotiations and tradeoffs between business partners having incompatible (heterogeneous) data and services. The processes should take into account data models of the resources, but first of all the global canonical schema is influenced by the business model required by global applications (operating on top of a grid). Therefore a development of an object-relational wrapper for a grid is 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 backend should work on a given (preferably arbitrary) relational database.

The major problem with this architecture concerns how to utilize a powerful native SQL optimizer. In all known RDBMSs the optimizer and its particular structures (e.g. indices) are transparent to the SQL users. A naive implementation of a wrapper causes generation of primitive SQL queries such as *select * from R*, and then, processing the results of such queries by SQL cursors. Hence, the SQL optimizer has no chances to work. Furthermore, such an approach causes large amounts of excessive data to be retrieved from a relational resource (bandwidth limitations, many IO operations) and then further processed. It is exceptionally inefficient. Our experience has shown that direct translation of object-oriented queries into SQL is infeasible for a sufficiently general case.

The solution to this problem presented in this paper is based on the object-oriented query language SBQL [1], virtual object-oriented views defined in SBQL, query modification methods [3], and a mechanism capable of detecting in a query syntactic tree patterns that can be directly translated to SQL-optimizable queries. Such patterns match typical optimization methods that are used by SQL query optimizers, in particular, indices and fast joins. Relatively small partial query results returned from a relational database are then evaluated in the whole query context (resulting from an already SBQL-optimized query form). The concept of the two-stage optimization (object-oriented and relational) is original and innovatory.

The rest of the paper is organized as follows. In Section 2 we present a brief state of art and the fundamental concepts of the described solution including SBA, SBQL and updateable object views. Section 3 presents in details the data grid architecture. Section 4 discusses an object-relational wrapper, its action and an optimization procedure with an example. Section 5 concludes.

2. Motivation and Fundamentals of the Solution

Object-oriented wrappers build on top of relational database systems date to late 1980s and were developed with federated databases. Their motivation was a reduction of technical and cultural differences between traditional relational databases and novel object-oriented paradigms. Recently, Web technologies based on XML/RDF also require similar wrappers. Although object-oriented and XML-oriented technologies are rapidly evolving and offering new fields of application, people are accustomed and quite satisfied with relational databases and there is a little probability that the market changes soon to other data store paradigms. Furthermore, the market is saturated with (object-) relational DBMS-s and costs of a software replacement and a data migration would be incredibly high and software companies are not willing to develop and introduce new technologies and reject existing ones, as a large demand for them still continues.

A database "tower of Babel" makes a database programming constrained to a particular DBMS. Even flexible technologies like JDBC cannot cover SQL irregularities and object-relational wrappers (e.g. Torque [4], OJB [5], Hibernate [6]) are not efficient, especially in a grid case. Hardly do they allow effective optimization (due to programming language runtime objects to SQL operations transformation), either.

A grid integration requires a generic and effective data model mapping. The 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 database objects. In our opinion, the Stack-Based Approach and its query language SBQL [1] 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.

Currently, our team is working on a data grid solution developed under international eGov-Bus project (contract no. FP6-IST-4-026727-STP). The

project objective is to research, design and develop technology innovations which will create and support a software environment providing user-friendly, advanced interfaces supporting "life events" of citizen or enterprises – administration interactions transparently involving many government organizations within the European Community [7]. This objective can be accomplished only if all existing government and para-government database resources (heterogeneous and redundant) are accessible as a homogeneous data grid.

It will be based on our own object-oriented query language SBQL, having a precise formal semantics, which is a prerequisite for developing any automatic transformations of queries into semantically equivalent forms. SBQL is already implemented, including its typechecker and a query rewriting optimizer. Furthermore, 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 (partially implemented [9]). There are three basic applications of the views:

- as integrators (mediators) making up a global virtual data and service store on top of distributed, heterogeneous and redundant resources,
- as wrappers on top of particular local resources,
- as customization and security facility on top of the global virtual store.

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. Moreover, because SBQL views are stateful, have side effects and be connected to classes, one would be able to implement a behaviour related to the objects on the SBQL side.

The major problem concerns how to utilize the SQL optimizer. Our experience leads to a conclusion that static (compile time) mapping of SBQL queries into SQL is infeasible. 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 backend 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, for instance, pushing selections before joins,
- indices, i.e. internal auxiliary structures for a fast access,
- fast joins, e.g. hash joins.

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 [3], 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 sub-trees we issue SQL execute immediately statements.

2.1. The Stack-Based Approach and Updateable 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 (ENVS). SBA extends this concept for the case of query operators, such as selection, projection/navigation, join, quantifiers and others. 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 family 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 [15] and XML-oriented query languages such as XQuery [16]. SBQL, together with imperative extensions and abstractions, has the computational power of programming languages, similarly to Oracle PL/SQL or SQL-99.

SBA assumes the object relativism principle that makes no conceptual distinction between objects of different kinds or stored on different object hierarchy levels. Everything (e.g. a Person object, a salary attribute, a procedure returning the age of a person, a view returning well-paid employees, etc.) is an object. SBQL respects the naming-scoping-binding principle: each name occurring in a query is bound to the appropriate run-time entity (an object, an

attribute, a method, a parameter, etc.) according to the scope of its name. The principle is supported by means of the environment stack (ENVS). The concept of the stack is extended to cover database collections and all typical query operators occurring, e.g. in SQL and OQL.

Due to the stack-based semantics, the full orthogonality and compositionality of query operators have been achieved. The stack also supports recursion and parameters: all functions, procedures, methods and views defined in SBQL can be recursive by definition. 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 [8].

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 [10, 11, 12].

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 [10, 11, 12].

3. A Data Grid Architecture

The main idea of a data grid is based on a virtual repository for database services [6]. A virtual repository and the basic mechanisms of a grid are shown in Fig. 1.

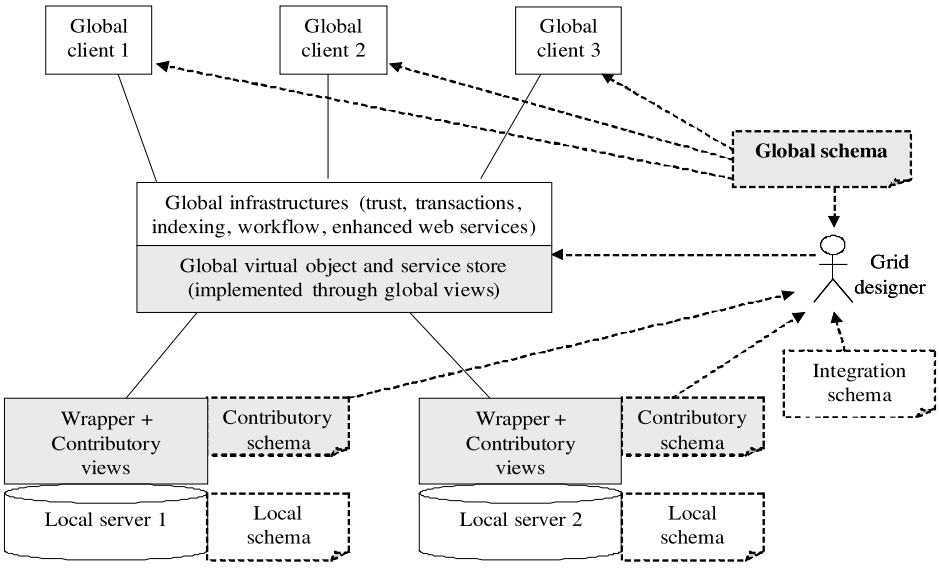

Fig. 1: Architecture of a data grid

A virtual repository is an interface to distributed data residing at each local data provider. The general goal of virtual repositories is a simplification of an access to distributed, heterogeneous and redundant data. Our goals are: designing a platform where all clients and providers get access to thousands of distributed resources without any complications of a data maintenance, building a global schema of accessible data resources, designing a transport platform for a free data and information exchange. A big advantage of our conception is that data and services need not be copied, replicated and maintained at a global applications' side (in a global schema) – they live at their autonomous sites and are locally supplied, stored, processed and maintained [13].

Further simplifications can be achieved with additional mechanisms contained within our main virtual repository using an object-oriented database with the Stack-Based Approach (SBA) [1].

The principal problem is how to join local clients and resource providers as parts of a global virtual repository with free bidirectional data processing. Our conception defines wrappers and adapters as media which can transport available data inside a grid. These mechanisms exploit other services available at resource providers and build an access bridge between an original structured

content and our second principal mechanisms called mediators and views. These mechanisms are responsible for a data formalization of unstructured content for a local schema which is a well formalized part of a global virtual repository. A local schema is a programmatic expression providing bindings between a formalized data representation and an original data service. It means that a local schema is only in a virtual and non-materialized state. As a result we get collections of business objects (fragmented or not) without redundancies, but with a direct access to proper data/service replications. A global schema is a principal mechanism which envelopes all local resources into one global data structure. Physically, it is a composition of contribution schemata which can participate in a grid. A global schema is responsible for managing grid contents through access permissions, discovering data and resources, controlling location of resources, indexing whole grid attributes. A global view is responsible for mapping data from local resources into a global schema. Such mappings consist of enclosing into a global schema particular contribution schemata residing at local sites, created earlier by local participants (resource providers). As a global client we understand a software implementation at a global client side. It enables accessing a global repository according to a trust infrastructure including security, privacy, licensing and non-repudiation issues. Global infrastructures and a global virtual object and service store are a collection of routines defining a whole virtual network infrastructure, covering physical network mechanisms compatibly with some trust infrastructure issues. A virtual network simplification relies on a peer-to-peer technology. A grid designer represents a person or a software team or a consortium determining primary dependencies and attributes of a grid. At the beginning they create a global schema which later can be contributed during a participation of clients and providers. They also define a metabase structure. Next, a grid designer defines a main contribution schema and an integration schema according to a grid data structure intention. A grid designer creates also a contributory schema. It represents the main rules for a data formalization schema for any local resource. Basing on this, local resource providers create their own contributory schemata adapted to an unique data structure present at their local sites. A contributory schema is generated for each resource by a local resource administrator. It contains formalization rules based on a contributory schema with an adaptation for a data structure and services accessed from a local site. A resource formalization represented in such a form becomes a part of a global schema. A contributory view is a query language definition for mapping a local schema to a contribution schema. A well defined contributory view becomes a part of a global view. The very important element of a virtual repository is an integration schema. It contains additional information about a method of resource integration into a grid, e.g. how to merge inside a virtual repository fragmented relational data structure where some parts of them are placed in separated resources. A grid designer must be aware of fragmentation issues, which is unnecessary for a local sites administrator. A wrapper is a mechanism for importing and exporting data between two different data models, e.g. our object-oriented grid solution at one

side and a relational data structure on the other side. A representation of data and a services' composition for each of grid participants is a local server, where a local schema describes a local resource data model and can be presented in an original data structure (e.g. relational, XML, HTML). It cannot be directly connected to a virtual repository.

The presented architecture of a data grid is fully scalable, as growing and reducing grid contents is dependent on a state of global schemata and views.

The approach to a data grid defines a query language as a main mechanism with a high-level access to original data covered behind a virtual repository. It specifies a conceptual, declarative, macroscopic and free from physical details access to distributed, heterogeneous and redundant data available in the whole grid [10, 11].

The approach should specify a security, privacy and non-repudiation infrastructure built on top of a virtual repository. It should be independent of clients' security and privacy models and designed for a grid transaction processing model. Parts of these aspects are covered within our transport platform.

3.1. A Virtual Repository Concept

The solutions exploiting virtual repositories currently available in the field implement particular solutions hard to be reused for other organizations and business goals. Processing in a virtual repository comprises some complex issues. One of them is updating virtual data seen through a virtual repository, the state of research about this problem is open, practically unexplored. Similar problems concern security and global transaction infrastructures that are built over of a virtual repository. Another problem concerns performance issues, in particular, a global query optimization concerning our object oriented database query language as a user interface for a virtual repository.

Problems described above as a result raise an issue of developing generic methodologies, environments, tools and languages that support a quick development of a grid with a virtual repository aiming at a particular application integration goal. This problem can be solved with an architectural idea of the grid components presented in Fig. 1 and with developing concrete technical solutions concerning particular components:

- developing a canonical model and a schema according to global user requirements (several directives like a business contract, a standard, law regulations, etc.). The model and the schema should be implemented in a corresponding language having both human and machine interpretations,
- developing local models and schemata of providers for participating local data and service resources in terms of a canonical model and a schema, i.e. showing how particular local providers contribute to the global schema,
- developing assumptions concerning export wrappers for particular providers sharing their resources,

- developing an integration schema exploiting a local schema and a global schema which shows dependencies between local resource providers and the global view and dependencies between the local providers (redundancies, replications, etc.).

The principal development and implementation goal is proper defining a global integrator of the virtual repository, which maps all local data and resources to the global view.

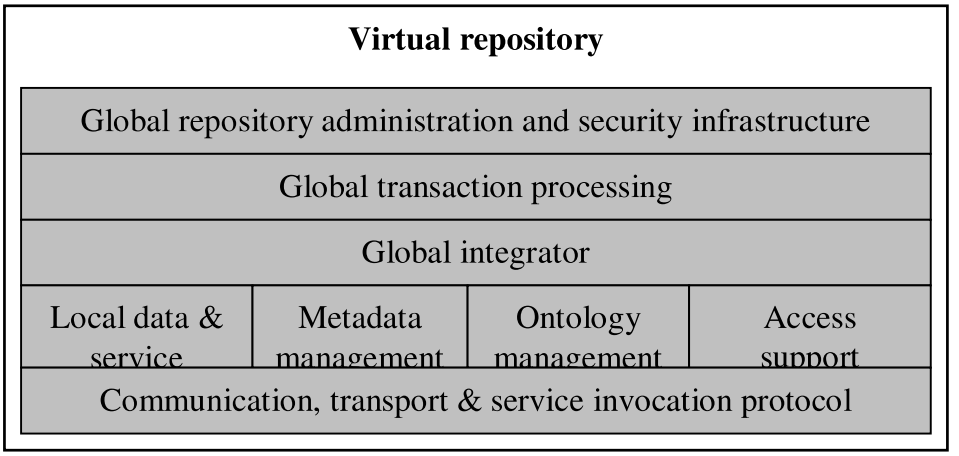

Fig. 2: Virtual repository

Refined vision of the virtual repository node is presented in Fig. 2. It includes other important architectural components:

- a global repository administration and a security infrastructure built on top of global virtual data and services, it is responsible for an administration of global resources and granting privileges to particular clients and resource providers,
- a global transaction processing is responsible for a synchronization of a concurrent access to distributed resources,
- a global integrator prepares a fragmented data integration into virtual wholes, resolves heterogeneities into reusable schemata, binds redundancies and replications for a virtual whole, employs access support mechanisms, and does other actions necessary to achieve assumed transparency forms,
- a metadata management keeps global and local schemata and interdependencies between local and global resources and between particular local resources,
- an ontology management keeps all user-oriented meta-resources concerning a classification, a categorization, dictionaries, a topic and knowledge maps, etc. The meta-resources have direct associations with the resources, allowing users to discover, retrieve and process them according to some conceptual patterns,

- an access support management consists of indices and query optimization mechanisms that support a performance, an availability and a scalability.

Some concrete technical goals, which include many issues, are implemented in the currently developed object oriented platform ODRA for web and grid applications which is the leading mechanism in our development:

- developing a powerful object-oriented database model able to cover all aspects occurring during an integration of heterogeneous resources having various data formats. The model will be supported by a schema language that can be used to specify a global schema and local schemata of participating resources,
- developing a powerful object-oriented query language, as API for processing resources,
- developing a mechanism of updateable object-oriented views used for several purposes: as global integrators, as schema's organizers on top of local servers, as first-class interfaces being the subject of administrative decisions concerning user privileges, and as an additional security mechanism based on overloading generic operations on virtual objects and supporting security policy changes,
- developing a communication, transport and service invocation protocol that would physically integrate resource providers with a virtual repository corresponding to other platform for a data transportation,
- developing a methodology disciplining the processes of manufacturing a virtual repository for a given business goal with well defined, measurable and controllable steps.

Although the literature contains many works concerning the above issues and problems, the field is rather in a premature stage, far from a complex and universal solution. Our research based on generic object-oriented database model with updateable views focused on unifying query languages and updateable views to design well defined grid mechanisms creates a big chance to receive significant theoretical and practical results much beyond the current state of art.

The technical aspects of realization of the described idea assume an existence of several cooperating technologies nearby. The principal aspect is a good design of a virtual repository platform corresponding with two other aspects which are responsible for data resources maintenance (wrappers/adapters) and a business data exchange including a grid security and its management (a transport platform). The concept concerns existing individual software modules with interconnectivity mechanisms enabling a restricted and specified participation of data processing. It has an original and general tendency. The modules may be developed and implemented according to the idea of a project's architect.

Our concept assumes the following strategies:

- a virtual repository – physically available with applications based on the previously described architecture. Parts of a repository will be a client and a provider application and a management application.
- import/export adapters and wrappers – mechanisms supporting a grid architecture to import and export local resources which may contribute to a virtual repository. They are software modules enabling a resource provider's services exploitation. Each of grid clients and providers will be equipped with user selected or corresponding to a grid contribution schema modules, which can discover local data and thanks to the views mechanisms grant access as a part of a virtual repository [11].
- a transport platform [13] – determines independent software environment responsible for free distributed transaction processing. The platform particularly should grant an unlimited physical access to a grid network for clients and resource providers (units) and an assurance of a well formed protocol for an information interchange. It is based on a centrally managed peer-to-peer network infrastructure. The following P2P features should assure operations such as: unit unique identifying, unit naming, units' interconnections, a network security, etc. Other important aspects are keeping a resources location transparency for acting units, a scalability of a network, an independence of a physical network configuration and naming. All these aspects can be developed using a multiprotocol, fully programmable P2P platform of the JXTA project [14].

3.2. A Grid Approach and a Peer-to-Peer Network

The principal technical difference between P2P and grid technologies is located in the field of data resources. In P2P, network processing consist of a defined in advance data exchange where the same physically data come from multiply resources. If more resources contain the same data then clients can easily contribute to this data. An accessibility of different data is the most important for a grid technology. There is a lot of data containing different information or similar information differently structured or combined. Looking at processing of business information, grid organizations need more data information accessible by an individual resource rather than more of the same data information better accessible by multiply resources. This attribute is bound with a global schema represented by data managed in a virtual repository in grid networks. They contain all formalized content accessible inside a grid and come from individual resource providers. P2P networks do not need a global schema because their content is imposed by an order of a superior authority.

In the presented concept P2P networking can be used as a grid transport platform for services and data interchange and a security layer.

4. A Wrapper Architecture and an Optimization Procedure

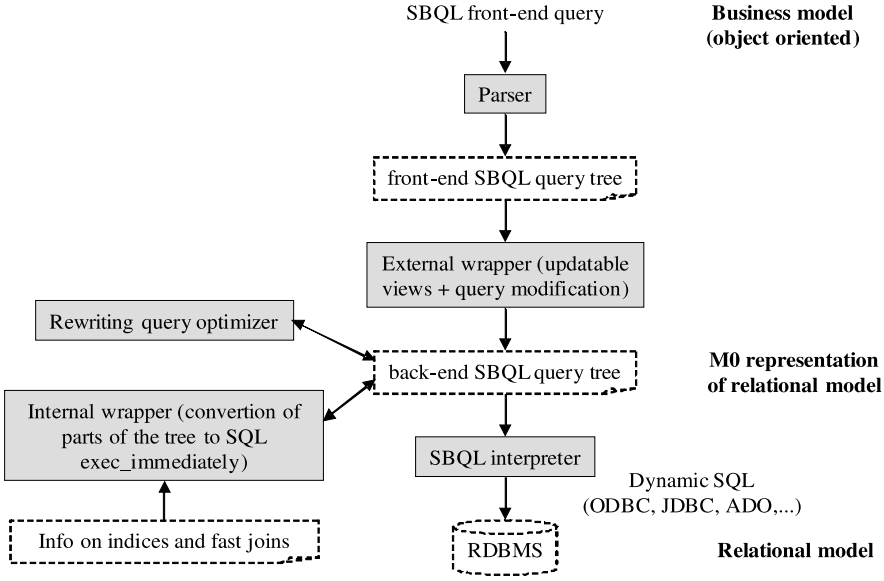

Fig. 3: Wrapper architecture

Fig. 3 presents the architecture of the wrapper. The general assumptions are the following:

- externally the data are designed according to the object-oriented model and the business intention of the global schema – the frontend of the wrapper (SBQL),
- internally the relational structures are presented in the M0 model (excluding pointers and nesting levels above 2) [11] – the backend of the wrapper (SBQL),
- the mappings between frontend and backend is defined with updatable object views. They role is to map backend into frontend for querying and frontend onto backend 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.

In other words, the wrapper is a middleware between the top SBQL engine and the bottom (resource) relational engine.

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 parametrized by queries, the major optimizations

concern just these parameters, with the use of the query modification technique as well. There are technical problems with identification of relational tuple within the SBQL engine (and further in SQL) for updating operations. 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 Fig. 3 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 catalogues. Then, the developers are forced to provide an utility allowing the wrapper designer to introduce this information manually. As stated in section 4, our solution is aware of this situation.

The query optimization procedure (looking from wrapper's frontend to backend) for the proposed solution can be divided into several steps:

1. A 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 backend.
2. The query is rewritten according to static optimization methods defined for SBQL [3] 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 analyse 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.

As a short example consider an SBQL query: `R where A = v`. If there is a SQL index on A column in R relation in the relational database, it is substituted (in the syntax tree) with *exec_immediately* clause invoking appropriate SQL query: *exec_immediately*(`"select * from R where A = v"`). Similarly, the pattern representing *primary-foreign key* can be found, the SBQL subquery can be substituted with *exec_immediately*(`"select * from R1, R2 where R1.PrimaryKey = R2.ForeignKey"`).

4.1 An Integration and Optimization Example

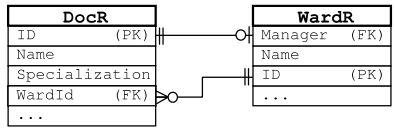

Fig. 4: Example of a relational schema

As the optimization example consider a simple two-table relational database containing information about doctors `DocR` and wards `WardR`, "R" stands for "relational" to increase the clearness (Fig. 4).

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

```
   create view DocDef {
      virtual_objects Doc {return DocR as d;}
      virtual_objects Doc(DocId) {return (DocR where ID == DocId)
   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.specialization as s;}
         on_retrieve {return s;}
      }
      create view worksInDef {
         virtual_pointers worksIn {return d.wardID as wi;}
         on_navigate {return Ward(wi) as Ward;}
      }
   }
   create view WardDef {
      virtual_objects Ward {return WardR as w;}
      virtual_objects Ward(WardId) {return (WardR where ID ==
   WardId) 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.menagerID as b;}
         on_navigate {return Doc(b) as Doc;}
      }
}
```

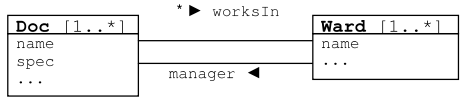

Fig. 5: Example of a corresponding object-oriented schema (wrapper's front-end)

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 employee 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 (relational model) available to the wrapper that can be used during the query optimization is that name column is uniquely indexed in either relation and there is a primary-foreign key integrity between *WardId* column (*DocR* table) and *ID* column (*WardR* table).

The optimization procedure is performed in the following steps:

1. Introduce implicit `deref` function
```
((Doc where worksIn.Ward.deref(name) = "cardiac surgery") where
deref(spec) = (Doc where deref(name) =
"Smith").deref(spec)).deref(name);
```
2. Substitute `deref` with the invocation of `on_retrieve` function for virtual objects and `on_navigate` for virtual pointers
```
((Doc where worksIn.(Ward(wi) as w).Ward.(name.n) = "cardiac
surgery") where (spec.s) = (Doc where (name.n) =
"Smith").(spec.s)).(name.n);
```
3. Substitute all view invocations with the queries from *sack* definitions
```
(((DocR as d) where ((d.wardID as wi).(((WardR where ID == wi)
as w) as Ward)).Ward.((w.name as n).n) = "cardiac surgery")
where ((d.spec as s).s) = ((DocR as d) where ((d.name as n).n)
= "Smith").((d.spec as s).s)).((d.name as n).n);
```
4. Remove auxiliary names `s` and `n`
```
(((DocR as d) where ((d.wardID as wi).(((WardR where ID = wi)
as w) as Ward)).Ward.(w.name) = "cardiac surgery") where
(d.spec) = ((DocR as d) where (d.name) =
"Smith").(d.spec)).(d.name);
```
5. Remove auxiliary names `d` and `w`
```
((DocR where ((wardID as wi).((WardR where ID = wi) as
Ward)).Ward.name = "cardiac surgery") where spec = (DocR where
name = "Smith").spec).name;
```
6. Remove auxiliary names `wi` and `Ward`
```
((DocR where (WardR where ID = wardID).name = "cardiac
surgery") where spec = (DocR where name = "Smith").spec).name;
```
7. Now take common part before loop to prevent multiple evaluation of a query calculating salary value for the doctor named *Smith*
```
((((DocR where name = "Smith").spec) group as s).(DocR where
((WardR where ID == wardID).name = "cardiac surgery")) where
spec = s).name;
```
8. Connect `where` and navigation clause into one `where` connected with `and` operator
```
((((DocR where name = "Smith").spec) group as s).(DocR where
(WardR where (ID = wardID and name = "cardiac surgery")) where
spec = s).name;
```

9. Because `name` column is uniquely indexed (in `DocR`), the sub-query `(DocR where name = "Smith")` can be substituted with *exec_ immediately* clause
```
(((exec_immediately("SELECT specialization FROM DocR WHERE name = 'Smith'")) group as s).(DocR where (WardR where (ID = wardID and name = "cardiac surgery")) where spec = s).name;
```
10. Because the integrity constraint with `DocR.WardId` column and `WardR.ID` column is available to the wrapper (together with information about the index on `WardR.Name`), the pattern is detected and another *exec_immediately* substitution is performed:
```
(((exec_immediately("SELECT specialization FROM DocR WHERE name = 'Smith'")) group as s).(exec_immediately("SELECT * FROM DocR d, WardR w WHERE d.wardID = w.ID AND w.name = 'cardiac surgery'") where spec = s).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).

5. Conclusions

The presented approach to data grid concerning wrapping relational databases to object-oriented business model with application of the stack-based approach and updatable views is clear and implementable. A frontend SBQL query can be modified and optimized with application of SBA rules and methods within the wrapper (updatable views) and then powerful native relational optimizers for SQL language can be employed. The amounts of data subsequently processed by the wrapper are satisfactorily small.

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 (which can be read directly from the relational metadata or manually entered in the wrapper's schema if not available directly). The SQL optimization is out of the scope of the wrapper action and is assumed to be efficient and reliable.

We have also elaborated a similar solution for semistructured data (XML) basing on Lore system.

References

[1] Subieta K.: *Teoria i konstrukcja obiektowych języków zapytań*. Wydawnictwo PJWSTK, Warszawa 2004, (522 strony).

[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] Subieta K., Plodzien J.: *Object Views and Query Modification.* (in) Databases and Information Systems (eds. J. Barzdins, A. Caplinskas), Kluwer Academic Publishers, 2001, s. 3-14
[4] Torque, http://db.apache.org/torque/
[5] OJB, http://db.apache.org/ojb/
[6] Hibernate, http://www.hibernate.org/
[7] eGov-Bus, http://www.egov-bus.org
[8] Plodzien J.: *Optimization Methods in Object Query Languages*, PhD Thesis. IPIPAN, Warszawa 2000
[9] Kozankiewicz H., Subieta K.: *SBQL Views - Prototype of Updateable Views*. ADBIS 2004
[10] Kozankiewicz H., Stencel K., Subieta K.: *Implementation of Federated Databases through Updateable Views*. Proc. EGC 2005 - European Grid Conference, Springer LNCS, 2005.
[11] Kozankiewicz H., Stencel K., Subieta K.: *Integration of Heterogeneous Resources through Updatable Views*. ETNGRID2004 WETICE2004, Proceedings published by IEEE.
[12] Kaczmarski K., Habela P., Subieta K.: *Metadata in a Data Grid Construction*. Proc. of 13th IEEE WETICE-2004, Italy, June, 2004.
[13] Kuliberda K., Kaczmarski K., Adamus R., Błaszczyk P., Balcerzak G., Subieta K.: *Virtual Repository Supporting Integration of Pluginable Resources*. 17th DEXA 2006 and 2nd International Workshop on Data Management in Global Data Repositories (GRep) 2006, Proc. in IEEE Computer Society
[14] Project JXTA Community: http://www.jxta.org
[15] Object Data Management Group: *The Object Database Standard ODMG, Release 3.0*. R.G.G.Cattel, D.K.Barry, Ed., Morgan Kaufmann, 2000
[16] W3C: XQuery 1.0: *An XML Query Language. W3C Working Draft 12*, November 2003, http://www.w3.org/TR/xquery/