Efficient Virtual Data Integration Based on XML

Konstantin Antipin[1], Andrey Fomichev[2], Maxim Grinev[2], Sergey Kuznetsov[1], Leonid Novak[1], Peter Pleshachkov[2], Maria Rekouts[2], and Denis Shiryaev[1]

[1] Institute for System Programming of Russia Academy of Sciences, B. Kommunisticheskaya, 25, Moscow 109004, Russia
{stakv, kuzloc, novak, shiryaev}@ispras.ru
[2] Moscow State University, Vorob'evy Gory, Moscow 119992, Russia
{fomichev, grinev, pleshachkov, rekouts}@ispras.ru

Abstract. In the scope of increased interest to the problem of integrating disparate heterogeneous data sources, virtual approach seems to be quite perspective and promising. In the general formulation this problem is extremely difficult to solve and has paid little attention so far. However the rapid evolution of XML – the multipurpose format for data representation – and elaboration of XML data query languages such as XQuery gives a new outlook on the old problem.
This paper contains a description of the overall architecture and foundations of BizQuery[3] – the virtual integration system based on XML data model. The system maps local sources as views on the global scheme and allows users to query data in terms of XML and UML via declarative languages XQuery and UQL in a uniform way. The problems of scheme mapping, query optimization, decomposition and processing in the case of virtual integration are touched upon.

1 Introduction

The task of heterogeneous data integration is among the basic and oldest problems related to data management research area. The concise formulation of the problem is as follows. Assume there are a number of heterogeneous data sources logically interrelated in some way. It is required to provide some software means for unified access to that data in such a manner as if they had a single logical and physical representation. We make no attempt to prove the significance of this problem, because we believe it is obvious.

In general there exist two basic approaches to this problem. The first approach consists in construction of warehouse snapshots of data. With this technique data subject to integration is being transformed in accordance with target integration model and placed into single warehouse database. A great deal of descriptive material concerning this integration technique is widely available, a state of art review could be found at [1].

[3] This work was partially supported by the grant 02-07-90300-v of RFBR.

The second approach is based on the virtual integration paradigm, when data from disparate sources is not materialized inside integration system, rather the on-the-fly query processing technique is used whereas user queries to integrated data are transformed into subqueries to separate data sources and eventually the data integration system forms the request results. Brief review on the evolution of this kind of systems, including multi-databases [2] and federated databases [3] can be found in [4]. The specific of exactly these systems is that they are targeted to integration of structured data. Later the mediator-based [5] integration systems appeared, which became mainly oriented on semistructured data model [6]. The emergence of XML [7] and attendant technologies (XSLT [8], XQuery [9]) induced a new wave of research activities around virtual data integration [10], [11], etc.

This paper discusses virtual integration system BizQuery, which is based on XML [7] and UML [12] technologies, and presents some results of research activities carried out by our R&D team during nearly a three years research project on heterogeneous data integration. Some basic characteristics of BizQuery are as follows:

- integrated access to several separate data sources, which can be relational or XML;
- the use of XML both for internal data representation and representation of query results;
- representation of the global scheme of the integrated data both in terms of UML and XML;
- the use of declarative query languages UQL (which is developed by ourselves [4]) and XQuery for querying integrated data in terms of UML and XML respectively;
- the full-featured query processing, including query optimization, query decomposition into the partial queries to individual data sources, and composition of the final result with potential joining and transforming of data.

The primary goal of the BizQuery project was to research problematic virtual integration approach and to develop implementation of a virtual integration system for practical use on the basis of XML family technologies (that is using XML as a native internal representation format). The authors tried to justify that virtual data integration approach could be quite realistic and practical in a number of recent problems of modern business. In the paper we consider the BizQuery system general architecture, as well as cornerstone concepts lying in the basis of BizQuery. We also strive to motivate our decisions on selecting corresponding techniques for solving data integration problem.

The remainder of the paper organized as follows. Section 2 describes the general architecture of BizQuery. Sections 3 and 4 penetrate into details of the two main system components – BizQuery Mapper and BizQuery Integration Server. Section 5 gives some performance results. Section 6 concludes the paper.

2 Architecture and System Structure

In this section we consider the general architecture of BizQuery and give the ideas lying behind main functional components. We would like to proceed from introducing the two phases of system operation:

- deployment phase, preliminary to system usage and responsible for constructing data integration meta-definitions (Fig. 1a);
- system run-time phase, when the system handles user queries to integrated data (Fig. 1b).

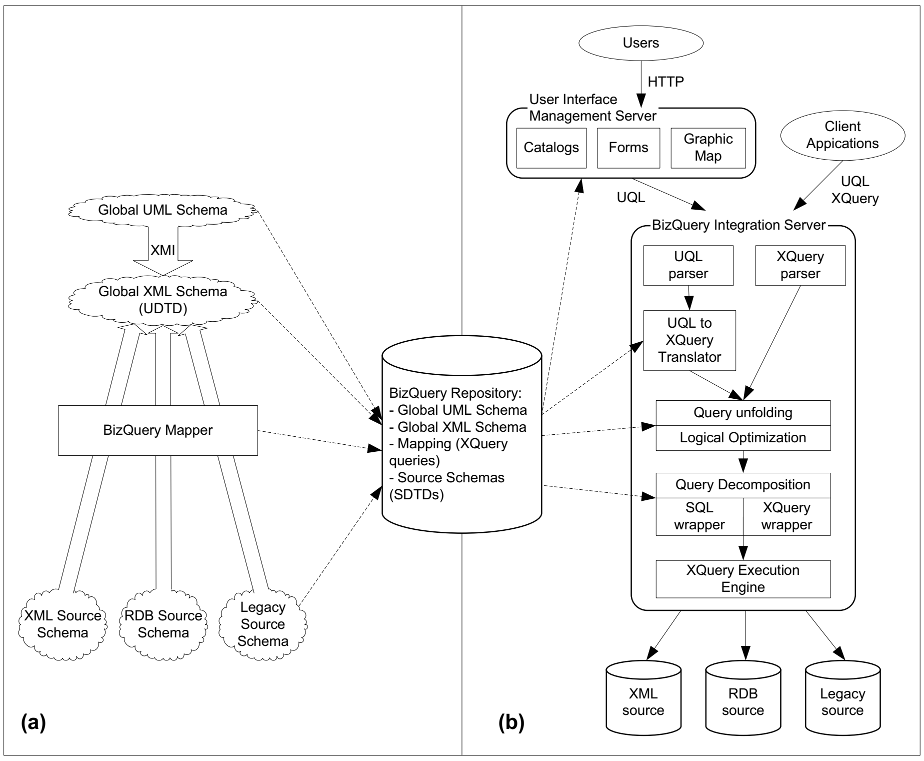

Fig. 1. System structure: (a) – deployment phase; (b) – run-time architecture

Before passing queries to integration system it is required to execute a number of preparatory activities aimed to define the system run-time configuration. Namely to define the global integrated scheme of data, either in the terms of XML or in the terms of UML; to gather and refine information on the schemes of data sources being integrated; to set the mapping of data sources schemes onto the global scheme. All of the enumerated tasks must be carried out during the BizQuery system deployment phase. The information collected during the deployment phase comes into BizQuery Repository, which serves as the configuration basis for BizQuery functioning at run-time phase. It is worth mentioning that during system deployment phase only meta-information is used (i.e. the schemes of data sources and global scheme), so no data is involved.

2.1 Deployment phase

BizQuery provides two possible levels of global scheme definition and access to integrated data:

- in terms of Global UML Scheme;
- in terms of Global XML Scheme.

In the full scenario the deployment process starts from the construction of the Global UML Scheme, which presents the given application domain. UML class diagram is used as a notation. When constructing this scheme, the two following aspects should be kept in mind. Firstly, the scheme must comply with the requirements of the users; secondly, the information adequacy of available or to-be available data sources to be integrated. The person who performs deployment of the system is responsible for thorough hitting both factors. At the next step, the specified Global UML Scheme is being automatically transformed into XML representation in accordance with the OMG XMI [13] specification. As a result we get Global XML Scheme. We would like to emphasis that the Global XML Scheme is the primary meta-informational resource for BizQuery run-time functioning, no matter how this scheme was constructed. In other words, if one intends to build data integration system right at the XML level, it is not required to create UML model first, if the final Global XML Scheme can be stated directly.

As mentioned above, at the deployment phase only data scheme information is used. Since we were interested basically in structural information about integrated data sources, the native XML DTD standard was accepted for representing data scheme information. Currently we work on implementing support for RelaxNG [14] data scheme encoding. Thus we need scheme information for each data source involved into integration process. Scheme information about XML data sources should be provided from that data sources directly, while for the sources of relational kind the scheme is built automatically from the internal catalog of corresponding RDBMS with predefined rules.

The final and most complicated step of deployment phase consists in building the mapping definition between the schemes of original data sources and Global XML Scheme. BizQuery provides a special mapping tool named BizQuery Mapper used to specify the denoted mapping statements, which will be discussed later in a separate section of this paper.

2.2 Run-Time Architecture

BizQuery run-time system consists of the two main components:

– BizQuery Integration Server (BQIS);
– User Interface Management Server (UIMS).

BQIS is responsible for processing queries formulated in UQL or XQuery. At that, every UQL query is being translated to XQuery by the UQL to XQuery Translator subsystem, and query processing is accomplished in the terms of XML. The availability of the metadata makes it possible since this metadata, on the one hand, describes the original model and, on the other hand, is represented in the XML itself.

The XQuery query is subject for query unfolding. Unfolding consists in substituting views stored in the BizQuery Repository (obtained at the deployment phase) into the query. Note that in the result of substitution the query is reformulated in terms of data source schemes, and its structure usually becomes more complicated. After that, the query is optimized by the Logical Optimizer applying rewriting rules, which can make significant simplification and "improvement" of the reformulated query. This step is one of the most important for the overall system performance and will be discussed thoroughly later.

Then optimized query is decomposed into partial subqueries (still in XQuery), each formulated in terms of the scheme corresponding data source (one or more), involved in the global query. Every partial subquery is then translated into the language comprehensible by corresponding data source. This task is being done with one of BizQuery wrappers. Currently BizQuery supports SQL and XQuery wrappers. At that, query translation for XQuery data source is trivial, while translation of arbitrary XQuery expression over relational data source into the SQL query is not a simple task. We will discuss this issue in more details later.

After decomposition phase the query is made up of a number of subqueries to the data sources and the so-called cross-source part (i.e. the part of the query, which take data from data sources as operands). The cross-source query part should be executed by the integration system, i.e. with XQuery Execution Engine.

BQIS module provides an open API for developing client applications, which can be used to issue XQuery and UQL queries to BizQuery integration system. However, this could not be considered as a convenient end-user interface for access to the integrated data because it requires essential programmatic effort. What we strived to provide with BizQuery was some user interface management component to fill the gap between BQIS and the end-user. That is what UIMS is. UIMS module provides automatic construction of three kinds of GUI interfaces for integrated data access in terms of UML.

Catalogs GUI provides navigational data access facilities. With the catalogs, the user browses integrated data as the instances of UML classes in correspondence with the UML model in focus. While navigating catalogs, the user may follow the typified links between the instances of UML classes, in accordance with the model definitions.

Another two branches of the GUI – Forms and Graphic Map – implement declarative query formulation technique. Basically, both of these GUIs provide facilities to travel around the UML model in focus, following the links between classes and setting constraints on the values of attributes at each step. In both cases the result of the user activity in all kinds of GUI is a UQL query composed by UIMS. This query will be send later to BQIS for execution. After the query is executed, UIMS passes the query result back to the client, providing adequate composing and formatting of the result. In all three scenarios the user operates in the terms of UML model and corresponding UQL queries are constructing automatically.

UIMS was implemented as a web server application and possesses rich customization capabilities due to the use of the XSLT processor, responsible for composing each component of user interface. GUI instances are being built automatically and dynamically on the basis of the UML model kept in BizQuery Repository.

2.3 Declarative Query Languages of BizQuery System

According to the concept of two integration layers – XML and UML – BizQuery provides two languages for querying integrated data namely XQuery and UQL. XQuery, being elaborated at the W3C consortium and being the most perspective declarative language for querying XML data, was accepted in BizQuery as a general internal language.

In BizQuery the user may define XQuery queries upon two principal kinds of entities:

- virtual documents that correspond to the global XML scheme. A keyword "virtual" inside function "document" is used for referencing (e.g. `document("virtual:foo.xml")`);
- real documents that are actually existing XML documents or tables of relational database (e.g. `document("real:sql/foo")`).

In the first case the user works in terms of the virtual document or view, which has the underlying query. The purpose of this query is to express construction of the virtual document in terms of real documents existing in the system. In the second case the user appeals directly to the entities of data sources (because all the internal processing of data is being done in terms of XML, contents of relational tables are trivially encoded in XML using a set of predefined rules).

While developing data manipulation techniques for BizQuery, we realized the necessity in a language for querying data in terms of UML model. Thus we have developed UML Query Language (UQL for short) that is a language used to formulate queries in terms of class instances of corresponding UML model that is the Global UML Scheme of BizQuery.

The UQL language is based on OCL [12] with a slight difference in syntax and semantics. The main purpose of OCL is to define constraints upon the classes and their attributes in correspondence with the UML model in focus.

Generally speaking, UQL is meant for querying instances of UML classes via imposing predicates on the attributes of classes, moving through the links between classes and the use of generality and existential quantifiers.

Here is an example of UQL query. The semantics of the query is as follows. Select all closed auctions, that are instances of the class open_auction, which have price more than 40 and the buyer is a person with income at least 50000 a year.

```
context model-id("1803"):
extent(closed_auction)=>
select(c|c.price>"40" and
         c!buyer@person=>exist(p|p.income>="50000"))
```

3 BizQuery Mapper

As we have stated it earlier, the basic task of the BizQuery deployment phase consists in constructing the mapping of integrated data on the data corresponding to the Global XML scheme. Generally speaking, the Global XML scheme may be of arbitrary complexity. Consequently, the mapping of data source onto the Global scheme may be quite non-trivial (i.e. the schemes may be subject to a sequence of complex transformations).

Before covering the basics of BizQuery Mapper component, responsible for solving the mapping task, let's make some classification of scheme mapping approaches. In common sense this task can be formulated as follows. Consider we have some data that conforms to the scheme A. How to transform this data to make it conformed to the scheme B? There are several methods that solve this problem:

1. **Transformation with a program.** We could develop a program in some general-purpose language (e.g. C++ or Java), which will make specifically this kind of data transformation from scheme A to scheme B.
2. **Manual transformation.** We could construct a request on some query language (say XQuery) that is to be applied to the original data to get the result conforming to the scheme B. This method, as well as all the subsequent ones, has a remarkable property: transformation defined in a declarative language can be a subject to optimization.
3. **Transformation in high-level terms.** We could shift from the terms of XML nodes to the terms of trees corresponding to the original and target scheme instances A and B correspondingly. When constructing transformation in this manner we operate with some tree algebra expressions, which can be automatically translated to query language expressions later and applied to the data in focus.
4. **Scheme matching.** With this method, we assume the presence of some initial algorithmic activity on searching for semantically corresponding nodes followed by more exact manual mapping corrections. As the previous method, this one should produce a query to be applied to the data in focus.
5. **Automatic high-level terms transformation.** With this method, we exploit the technique of method 3 yet trying to construct mapping definition automatically on the basis of semantic and statistic information on the data in focus.

6. **Automatic scheme matching.** With this method, we use the technique of the method 4 trying to construct the mapping definition automatically on the basis of semantic and statistic information on the data in focus.

In accordance with the classification given, BizQuery Mapper is predominantly based on the method 3 and partially uses the method 4 as well. Our objective was to provide a full set of functions to perform a scheme transformation and obtain an environment, which would be comprehensive to the end-user, because the set consists of high-level algebraic operations that are generic to data management (i.e. selection, projection, join, etc.). In some cases the optional scheme matching method algorithms may do the work of scheme transformation more easy and convenient.

Thus, we move from the scheme A to the scheme B iteratively via step by step application of transformation functions of BizQuery Mapper. The set of transformation operations is closed regarding the variety of schemes and consequently it is an algebra. The only constraint is the requirement for the resulting scheme to be inferable from the original one. In practice this means that there must be no dependencies on data in definitions of transformations. For example, it is illegal if the content of some element becomes the name of a new element.

Designing the set of transformation functions we were guided by the goal to provide convince of usage rather than to minimize the set. The transformation functions operate with scheme subtrees rather than nodes or node sequences of the XML document, so many queries could be specified much simpler than with XQuery. Nevertheless the mapping built in high-level terms is translated into XQuery micro operations, which are good for optimization and evaluation.

BizQuery Mapper, which provides this functionality, is implemented as a separate component with a convenient user-friendly graphical interface including drag-and-drop facilities.

4 BizQuery Integration Server

In this section we will consider the BizQuery BQIS component responsible for XQuery and UQL queries execution. It consists of the following functional parts: UQL and XQuery parsers, UQL to XQuery translator, query optimizer, query decomposition subsystem, query execution subsystem and finally wrapper subsystem providing connection with the data sources.

It is a well-known fact that a virtual integration approach has principal performance constraints, which may lead to unsatisfactory response times. These principle constraints result from the lack of actual data statistics and data structures (i.e. indexes) providing an optimized access to the data. Additionally, some problems originate from the delays on data transfers over the network as well as transformations of data into some internal representation for further processing. These difficulties follow from the fact that virtual integration system does not materialize the data being integrated.

The authors realize in full measure that virtual integration systems would hardly ever become capable to show performance comparable to warehouse-like systems, however they seem to be practicable for those business tasks requiring high actuality of data.

During our work on developing BizQuery Integration Server module we have marked out three general components of the server enumerated below in decreasing significance order:

1. Logical optimization based on query rewriting (unfolding and simplification);
2. Query decomposition (extraction of maximum subquery for a source);
3. Stream processing at the server side and at the level of sources if possible.

Let's consider each of these components in more details.

4.1 Logical optimization

It is hard to overestimate the importance of query optimizer for DBMS: the difference in execution times for optimized and non-optimized queries may constitute magnitudes of order. Many present industrial DBMSs actively use cost-based optimization techniques, making the choice of the optimal query plan on the basis of the operation cost. The optimal plan is recognized as a plan with a minimal summarized cost of operations. Another optimization method is based on a special rule set for heuristic-based equivalent query transformations leading to the construction of a new query being more effective than the original one. This optimization method is known as rule-based optimization or query rewriting. This is exactly the method used for optimization in BizQuery. This approach was accepted due to the following reasons:

– The lack of data statistics in the virtual integration system, which could be used to estimate costs of operations;
– Usually after the query unfolding the user query issued to the virtual document contains a plenty of redundant information, which can be significantly reduced using rewriting rules.

The later is of extreme significance since the size of the query may vary in orders of magnitude depending on the virtual document complexity. The following is the list of advantages achieved thanks to query rewriting in BizQuery system:

– elimination of "redundant" query constituents, arising during query unfolding;
– predicates push-down and elimination of "needless" constructor elements;
– rewriting the query to a more "declarative" representation (subquery to join translation);
– partial rewriting of recursive functions calls into non-recursive constructs (using scheme information), which are subject for further optimization;
– query reformulation into "well-aimed" form on the basis of schema information, which can eliminate superfluous data scans (for example, replacement of wildcard in a path query by certain XML element name).

An arbitrary query, which can be a view, may be rather complex. Particularly, this is due to the potential presence of recursive functions, which are typically used for implementing many transformational operations performing tree traversals. And this is exactly the case where optimizer plays the major role for effective functioning of the overall process. In practice a huge query of thousands of lines may be transformed into a several hundred lines query.

It is important to note that XQuery does not have explicit join operation, thus we had to introduce such logical and physical operation correspondingly. This led to emergence of the extended XQuery data model. Typically join is expressed via the FLWR construct and this obliges us to apply one strict execution algorithm – nested loop join. That is why subquery to join translation makes the query more declarative and allows using alternative, usually more effective, algorithms. However the presence of order in XQuery still makes queries less declarative, because it prohibits making arbitrary joins rearrangement.

Logical optimization is of great importance even more for the following reason. As a result of query rewriting we get a new query with quite an accurate structure. By convention, a query plan tree could be divided into three parts: in the leaves of the tree there are data selection operations with filtering predicates; the middle of the tree constitute join operations; the top part of the tree mainly contain data transformation operations (Fig. 2a). Such normalization of the query plays the key part for query decomposition on later stages, which we are going to discuss a bit later.

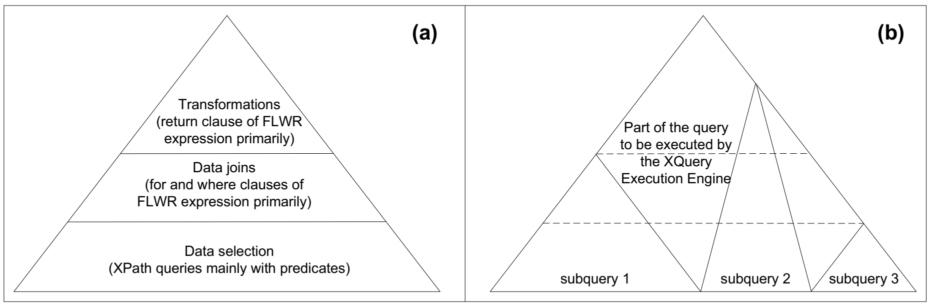

Fig. 2. XQuery query outline: (a) – after normalization; (b) – after query decomposition

The ideas lying behind the BizQuery optimizer are discussed in more details in [15].

4.2 Query decomposition

The query decomposition task consists in accomplishing query breakdown into a number of parts, aspiring to obtain such fragment of the whole query, which could be executed by data sources, while the rest binding part of the query to be executed by the integration system. Depending on the type of data sources subject to integration there exist two different cases. The first case regards data sources having extremely limited facilities for query formulation upon that data source, e.g. HTML-form interface [16]. Another case regards data sources with rather developed and powerful declarative means for declarative querying. As already mentioned above, in BizQuery we aimed to provide support for two kinds of sources - relational DBMS with SQL interface and data storage systems with XQuery interface. Thus the query decomposition task means choosing the "maximal" subqueries to data sources that can be passed to the data sources. For example, if in the original query contains such kinds of resource-consuming operations as joins or sorting regarding a single data source, this operations should be delegated to data source supposing that corresponding data source is capable to execute those operations itself. Here are the benefits of this approach:

– Execution of substantial parts of original query in parallel (due to parallel processing of subqueries by external servers);
– Qualitative data sources can execute queries faster than integration system due to the presence of extra meta-information about target data in their arsenal (e.g. index structures);
– For typical subqueries, amount of data, which must be transferred over the network, are usually much less compared to the size of the whole document kept at that data source (particularly, because of the presence of predicates in subqueries).

In that way, the task can be reduced to finding the maximal subtrees in the query execution plan tree, belonging to distinct data sources with subsequent translation of the extracted subtrees into a query language supported by corresponding data source. The rest of the global query keeps the cross-source operations, involving as operands data from several distinct data sources. And thus this part of the query is to be executed within the integration system (Fig. 2b).

It is necessary to note that it is extremely important for the query to be expressed in the normal form discussed above in order to accomplish subquery decomposition. As the leaves of the query tree are XPath expressions one can guarantee execution of subqueries by the data sources supporting corresponding query language. Higher in the tree, there are join and semijoin operations, which could be also passed to the data sources for execution if it is possible. In this way, we have to execute within the integration system only those join and semijoin operations, which are cross-source. In case of the query, which contains at least one cross-source operation before transformation operation, transformation processing is to be done within the integration system.

Let's give an example of query decomposition. Suppose we have a query referencing two different documents of the same relational data source constructing the sequence of departments with slightly modified structure and filtering employers whose age is below 20 years. Document schemes are presented in the table below as DTD.

```
for $d in document("real:sql1/deps")/table/tuple
```

```
where some $e in document("real:sql1/emps")/table/tuple[age<20]
      satisfies $d/id = $e/dep_id
return element dep {$d/name, element additional {$d/address}}
```

Table 1. DTDs for documents

document("real:sql1/deps")	document("real:sql1/emps")
<!ELEMENT table (tuple)*>	<!ELEMENT table (tuple)*>
<!ELEMENT tuple (id, name, address)>	<!ELEMENT tuple (name, age, dep_id)>
<!ELEMENT id (#PCDATA)>	<!ELEMENT name (#PCDATA)>
<!ELEMENT name (#PCDATA)>	<!ELEMENT age (#PCDATA)>
<!ELEMENT address (#PCDATA)>	<!ELEMENT dep_id (#PCDATA)\>

The query contains semijoin, defined in clauses "for" and "where", between two documents belonging to the same data source. This semijoin can be executed by the data source, while the operation on transformation figuring in the query is not supported by the data source. Shown below is the subquery to the data source sql1:

```
select * from deps
where (exists (select * from emps
               where deps.id = emps.dep_id and emps.age<20))
```

Someone can criticize the normal form of XQuery expression. Obviously there exist a number of simple transformations, which can be accomplished before join operations, for example vertical data projection at the side of relational source. However transformations of this kind are not resource consuming as denoted ones and could be executed within the integration system without degrade to system performance. Much more critical for achieving good performance is to bring to light "heavy" operations, i.e.joins and sorting.

4.3 XQuery Execution Engine

After the query decomposition phase the original query tree becomes transformed into the physical query execution plan with the subqueries in terms of data sources in the leave positions. At that the obtained query execution plan can be rather complex. This generally has the following reasons:

- the original query may include cross-source operations (e.g joins);
- the original query includes transformation operations, which are not supported by the corresponding data source (particularly this is the case when dealing with relational data sources).

Due to this reasons we need to have a full-functional XQuery processor within the integration system. Some operations upon extended XQuery data model, such as joins introduced above, are implemented as physical operations improving performance of the system.

We have adopted the iterative query execution model [17] widely used in relational DBMSs. This allows us to escape materializing intermediate processing results arising after execution of each operation. In fact, applying this approach for organizing query processing leads to a number of positive consequences.

We have discovered that the result of many user queries contains a sequence of XML elements (of the same type) rather than a single XML document. Particularly this holds for UQL queries producing as a result the sequence of class instances. Apparently, the scenario of consecutive user access to query results must be rather typical especially with the graphical web interface. Correspondingly it may be sufficient for the user to obtain some first portion of the whole result set with the possibility to obtain next portions gradually. This approach corresponds to the DBMS cursor idea allowing for essential reduce of query response time.

Strictly speaking with XQuery Execution Engine we made attempt to implement the so-called stream or pipeline processing techniques. Keeping in mind the fact that XQuery is functional language, it leads us to implementation of lazy semantics for XQuery. In fact implementing XQuery in lazy semantics contradicts to the XQuery specification, which defines XQuery as the language with static semantics. But at the same time the use of lazy semantics does not narrow the class of computable queries in any way. Indeed a query computable in the system employing static semantics model is also computable in the system with lazy semantics. Reverse is not true, that is some queries non-computable according to the XQuery specification can be computable with BizQuery.

5 Performance Study

Presented in this chapter are the results of BizQuery system performance measurement. It is important to note that to our best knowledge presently there exist neither recommendations nor facilities for testing and benchmarking systems like BizQuery. Thus we had to adopt such recommendations from XML DBMS society, namely XMark [18] benchmarking specification.

The document conforming to the XMark auctions DTD model was spited up into three distinct parts – one representing XML data source and other two representing relational data sources. As the server engine on the side of XML data source we have chosen QuiP [19] that is a XQuery processor developed by Software AG. The data for the QuiP processor resided on file system as XML files. Oracle 8i DBMS was used as server component at the side of each relational data source. Since QuiP is not a powerful DBMS it has limitations on amount of data processed. That is why the size of XML data rather small. In order to store relational part of benchmark in DBMS we had to make some adaptation of the corresponding part of the original XMark structure. Particularly we had to throw away structural nesting.

In the adopted benchmark scheme the first relational source consists of 5 tables with the total number of tuples more than 1.8 million. The second relational data source contains 3 tables with more than 4.2 million tuples in the whole. The XML data source consists of 4 files with the overall size of 5,8Mb. The total size of data files used to populate all the three data sources constitutes 700Mb.

The following hardware configuration was used to perform benchmark measuring. BizQuery Integration Server was run on computer with Pentium-IV 1500Mhz and 512Mb RAM. XML data source server (under QuiP) was started on the same computer with BQIS. Both Oracle servers were run on distinct computers with identical configuration - Pentium-III 733Mhz and 256Mb RAM. All programs were run under Windows 2000.

Table 2. Benchmark queries

Q1	for $x in document("real:sql1/item")/table/tuple
	where $x/QUANTITY="5" and $x/location="Germany"
	return $x
Q2	for $y in document("real:sql2/interest")/table/tuple
	for $z in document("real:sql2/people")/table/tuple[business="yes"
	and city="Moscow"]
	for $x in document("real:sql1/categories")/table/tuple[name="all"]
	where $y/ref_category=$x/id_category and $y/ref_person=$z/id_person
	return ($x, $z)
Q3	document("virtual:closed_auction.xml")
Q4	for $v in document("virtual:item.xml")/item[location="United States"]
	for $z in document("real:sql1/mailbox")/table/tuple
	where $v/id=$z/ref_item and $z/mail_date="12/11/99"
	return element {name($v)} {$v/*[not empty(./text())]}

The Table 2 shows queries, for which we publish measuring results. The query Q1 addressed the real document "item" mapped on the table of the first relational data source and just imposes the predicate. The query Q2 contains two joins. One join is defined between the documents of a single data source while another join is cross-source – defined between first and second relational data sources. The query Q3 demonstrates how to obtain a virtual document as a whole. Finally, the query Q4 expresses join between the virtual document "item.xml" and the real document "mailbox" from the second relational data source. Query execution characteristics are given in the Table 3.

As the reader can see, for all the four queries BizQuery processing time, spent on query optimization, processing of cross-source query parts and data transformations, is relatively small. This was achieved due to query rewriting (especially in Q4, joining virtual and real documents) and smart query decomposition shifting the burden of data processing to the data sources as much as possible. It is

Table 3. Results for benchmark queries

Query Number	Result size (in Kb)	Sources summary working time	BizQuery working time	Total query execution time
Q1	23	5,984	0,078	6,062
Q2	12	82,485	1,796	84,281
Q3	3673	67,907	3,093	71,000
Q4	27	52,766	0,938	53,704

worth mentioning that the total query execution time can be reduced via constructing indexes on the side of data sources, though this is out of the scope of this paper. Because of the paper size limits we cannot consider examples of logical query optimization allowing essential query normalization and simplification, which is vital for the queries with complex internal data transformations.

6 Summary and Conclusion

In this paper we presented the architecture of the BizQuery virtual data integration system based on the XML/XQuery data model, developed to provide integrated access to heterogeneous data in terms of XML and UML. We have considered the role of XQuery and UQL query languages. We have discussed methodology and tools for accomplishing mapping of integrated data sources schemes to the global scheme. The idea and technique of automatic interface generation on the basis of corresponding UML global model was introduced. Finally, we discussed the general performance problem of systems based on the virtual data integration approach and outlined the three main aspects of this problem, which the success of such systems depends on. We have also provided experimental results proving the efficiently of the presented techniques.

The problem of virtual data integration was and still remains the extremely difficult part of data management field. The emergence of the XML technology made it simple to represent all kinds of integrated data in a uniform way, but at the same time introduced the set of new problems including more complex query optimization and processing. During our work on BizQuery we made an attempt to bring to light these problems and to propose solutions for them. As a result, it has allowed us to build the efficient virtual data integration system and to prove that the virtual approach is quite practical.

References

1. A Selection of Papers on Datawarehousing, Computer, Vol. 14, No. 12 (2001)
2. Batini, C., Lenzerini, M., and Navathe, S.: A Comparative Analysis of Methodologies for Database Schema Integration, ACM Computer Surveys 18(4) (1986) 323-364

3. Sheth, A., Larson, J.: Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases, ACM Computing Surveys 22(3) (1990) 183-236
4. Grinev, M., Kuznetsov, S.: UQL: A Query Language on Integrated Data in Terms of UML, Programming and Computer Software, Vol. 28, No. 4 (2002) 189-196
5. Wiederhold, G.: Mediators in the Architecture of Future Information Systems, IEEE Computer 25(3) (1992) 38-49
6. Chawathe, S., Garcia-Molina, H., Hammer J., Ireland, K., Papakonstantinou, Y., Ullman, J., Widom, J.: The TSIMMIS Project: Integration of Heterogeneous Information Sources, IPSJ (1994) 7-18
7. Extensible Markup Language (XML) 1.0, W3C Recommendation, 2nd edition (2000) http://www.w3.org/TR/2000/REC-xml-20001006
8. XSL Transformations (XSLT) 2.0, W3C Working Draft 15 November 2002, http://www.w3.org/TR/2002/WD-xslt20-20021115/
9. XQuery 1.0: An XML Query Language, W3C Working Draft 15 November 2002, http://www.w3.org/TR/2002/WD-xquery-20021115/
10. The Tukwila Data Integration System, University of Washington, http://data.cs.washington.edu/integration/tukwila/
11. Xperanto Project, IBM Almaden Research Center, http://www.almaden.ibm.com/software/dm/Xperanto/index.shtml
12. Unified Modeling Language (UML), Specification Version 1.4, http://www.omg.org/technology/documents/formal/uml.htm
13. XML Metadata Interchange (XMI), Version 1.2 http://www.omg.org/technology/documents/formal/xmi.htm
14. RELAX NG Specification, Committee Specification 3 December 2001, http://www.oasis-open.org/committees/relax-ng/spec-20011203.html
15. Grinev, M., Kuznetsov S.: Towards an Exhaustive Set of Rewriting Rules for XQuery Optimization: BizQuery Experience, 6th East-European Conference on Advances in Databases and Information Systems (ADBIS), LNCS 2435 (2002) 340-345
16. Levy, A., Rajaraman, A., Ullman J. D.: Answering Queries Using Limited External Query Processors, PODS (1996) 227-237
17. Graefe, G.: Query Evaluation Techniques for Large Databases, ACM Computing Surveys 25(2) (1993) 73-170
18. XMark - An XML Benchmark Project, http://www.xml-benchmark.org
19. QuiP. Software AG's prototype of XQuery, http://developer.softwareag.com/tamino/quip/