Abstract—Efficiency in data manipulation is of vital importance to ERP systems. A flexible data manipulation application programming interface (API) helps to address a number of acute needs of application developers and eventually end users.

Building on the notions of business objects and Web services, this paper contributes the design and implementation of a business object query language (BOQL). Essentially, BOQL provides query-like service invocation API that on-the-fly orchestrates CRUD-operations of business objects in an ERP system. BOQL allows to achieve the desired level of data manipulation flexibility at a reasonable price. In order to improve the performance of BOQL queries they can be compiled. To demonstrate the feasibility of the suggested approach this paper presents three use cases for BOQL: composite application development, navigation in ERP data, and user interface configuration.

I. INTRODUCTION

One of the current trends in enterprise resource planning (ERP) systems development is turning ERP systems into platforms and allowing external applications accessing and manipulating systems data. This creates a need for a data manipulation application programming interface (API). A number of acute needs of ERP systems’ users and application developers cannot be fulfilled without flexibility in manipulating ERP data. Based on customer and developer feedback obtained from our industry partner (ERP software vendor), we concluded that ERP data organization and manipulation API are of vital importance in satisfying their urgent requirements: the customers struggle with poor navigation and customization capabilities of ERP systems, while the developers are constrained in manipulating ERP data by existing SQL-based API.

This paper elaborates on two questions: what makes an ERP data manipulation API efficient and how can this efficiency be achieved. The main contribution of the paper is a set of architectural and engineering guidelines on implementing a main-memory-based system (reflecting the current trend memory size growth) that addresses the urgent demands of customers and developers on data organization and manipulation. The significant part of the work has been dedicated to demonstrating feasibility of suggested approach (namely business object query language - BOQL), as it provides basis for a larger and more careful effort, including the in-depth development and analysis of formalism for BOQL (i.e. optimization algorithms, in-memory data layout, etc.). Essentially, BOQL provides query-like service invocation API that orchestrates at run time CRUD-operations of business objects. An option of compiling BOQL queries is also addressed in the paper. Hence, BOQL allows to achieve the desired level of data manipulation flexibility at a reasonable price.

The rest of the paper is structured as follows. Section II discusses the considerations that affected the design BOQL. Section III gives an overview of related work and analyzes the state of the art in ERP data access APIs. Section IV presents the main contribution of this paper, namely, business object query language. In particular, subsection IV-A describes the concept of BOQL, subsection IV-C discusses query compilation as a performance improvement option, subsection IV-D contributes with an implementation guidelines for an ERP system that exposes BOQL as an interface to its data, subsection IV-E addresses an important issue of exposing data model of an ERP system and how this must be done in order to enable BOQL. Section V presents three use cases for BOQL: composite application development, navigation in ERP data, and user interface configuration. Finally, section VI concludes the paper and outlines the future work.

II. DATA MANIPULATION API DESIGN CONSIDERATIONS

Before describing business object query language in details several considerations need to be mentioned as they strongly influenced the design of BOQL. The considerations are based on interviews with application developers of our industry partner, a leading ERP software provider with more than 40 years of experience in ERP systems development.

The first and the most significant aspect in manipulating ERP data is an ability to access any combination of attributes with as little effort as possible. ERP data models are often overwhelmingly complex due to extensive use of nested and hierarchical data structures. Almost always these structures get fragmented in order to be physically stored in a system. The fragmentation is a direct consequence of data normalization, the process of ensuring the maximum cohesion of entity types and eliminating redundancy in data models. The main argument in favor of normalization is that the management of normalized data is simpler than that of denormalized. The disadvantage of normalization is that it disjoints semantically coupled attributes of hierarchical and nested structures into a number of subsets and stores them separately (for example in different database tables). The data API in such systems (e.g. SAP R/3 and SAP Business Suite) is often organized around these subsets, in the sense that the system exposes operations for manipulating only the attributes.
that belong to a single table. This is very inefficient, because the semantics of data operations in most applications assumes manipulation on application domain objects (purchase order, customer invoice and etc.) rather than on database tables representing their parts. That is, applications need operations manipulating objects’ attributes at different levels of structural hierarchy of domain objects, whereas the system provides only operations manipulating attributes at the same hierarchy level. For example, a sales order header is stored separately from sales order items, and two different operations are provided to read them. In this case granularity provided by the API does not match the granularity needed by the application. Whenever such a mismatch occurs, application developers have to take the burden of expressing the required operation in terms of the ones available. This complicates the code and leads to situations, when data manipulation part of an application outweighs the application’s value adding part.

The second concern in ERP data manipulation is the assurance of the fact that any usage of the API does not compromise the integrity of ERP data. Manipulation of data must be performed in accordance with business logic rules. For example, when creating a sales order, availability check must be performed. Only after a successful confirmation a sales order entry can be created in the system. Without such a check, creating a sales order by simply inserting data into corresponding tables with SQL statements can result in a sale of not existing products.

The next consideration is compatibility assurance, that is the necessity of preventing invalidation of applications using the API in case of changes in data model of the system. Over their life-cycle, the data structures in ERP system may change. In this case, old client applications may suffer incompatibility with new versions of data structures. Ideally, a system must never reject calls from applications that successfully interacted with the previous versions of the system and should compensate the differences in data structures in order to preserve the integrity of the applications.

The forth consideration regarding ERP data manipulation is cross-platform support. The variety of platforms, from which applications access ERP systems, has dramatically increased over the last decade. Not respecting this fact may lead to incompatibility.

Last but not least, is the ease of use. Although obvious, this requirement is often overlooked. As consequence, powerful functionality can become unusable. In the context of ERP data manipulation the main usability factor in our opinion is the transparency of data model. Therefore, the target API must not only expose data of an ERP system, but also the system’s data model or metadata.

### III. RELATED WORK

A straightforward approach to data manipulation can be to use standard SQL. Since modern ERP systems rely on relational databases, SQL statements can be issued directly against the database tables in order to operate on data. Although feasible, this approach is unlikely to satisfy the requirements stated above. The problem with standard SQL is that it violates the data encapsulation principle. It exposes too much control over the underlying database and increases the risk of corrupting data in the system. An ERP system is not only data, but also a set of business rules that apply to the data. Generally, these rules are not a part of the system’s database. Direct SQL access to the data circumvents the rules and violates data integrity. Another disadvantage of SQL as API is the need for so called “glue code”: in order to manipulate data with SQL from the application layer the later must use a data provider (e.g. OBDC, ADO.NET, JDBC and etc.). This hinders the usability.

To increase the productivity of the previous approach and allow direct consumption of data from object-oriented programs, storage layers provide object views on existing relational databases [3]. The views are implemented on top of particular DBMSs to map persistent database relations to object models of applications. This approach has received the name of object-relational mapping (ORM) and is even more attractive with object-relational DBMSs, which support more of the desired object functionality in the database engine itself [4]. The advantage of ORM is having data objects as first-class citizens of a programming language. This simplifies coding and increases application developers’ productivity. A number of ORM generators can be used to alleviate the tedious development of mappings. Some development platforms even include such generators into their toolboxes. For example, the Enterprise JavaBeans specification offers two alternatives for defining the data access code of entity beans: Bean-Managed Persistence (BMP) and Container-Managed Persistence (CMP). In the later case a corresponding mapping is generated automatically by the bean’s container. The main disadvantage of mapping is low performance. Application object models are inherently navigational [4]. That is, objects have references or relationships to other objects, which applications follow one at a time. Each access to a relationship entails a round-trip to the RDBMS, which hinders the performance. Moreover, ORM results in replicating data in the application server. This in turn increases the consumption of hardware resources and leads to synchronization problems, when application resident objects need to be synchronized with data in the database.

An alternative to SQL can be data as a service approach. In this case a system exposes a number of Web services with strongly-typed interfaces operating on data. This approach has an advantage of hiding internal organization of data. Instead of a data schema and a query interface an ERP system exposes a set of operations that manipulate its data. By choosing operations and calling them in an appropriate sequence required actions can be performed. Because of using Web services this approach is platform independent. In fact, the data-as-a-service approach has been very popular. SAP, for instance, has defined hundreds of Web service operations that access data in SAP Business Suite. Amazon Electronic Commerce service is another example of such approach. However, this method has a serious disadvantage - lack of flexibility. Although an ERP system can expose many data manipulation operations,
they will unlikely cover all combinations of attributes that applications might need to operate on. Therefore, granularity mismatches are very likely to occur. As discussed earlier, this will require application developers to manually construct a sequence of calls on existing operations to perform a desired manipulation.

Service data objects (SDO) [5] enhance the data-as-a-service approach by specifying many aspects of data manipulation. SDO is a specification for a programming model that unifies data programming across heterogeneous data sources, provides robust support for common application patterns, and enables applications, tools, and frameworks to more easily query, view, bind, update, and introspect data [6]. SDO has a composable (as opposed to monolithic) architecture and is based upon the concept of disconnected data graphs. Under the disconnected data graphs pattern, a client application retrieves a data graph from a data source, mutates the it, and can then apply the data graph changes back to the data source. Access to data sources is provided by a class of components called data access services. A data access service (DAS) is responsible for querying data sources, creating graphs of data containing data objects, and applying changes to data graphs back to data sources. SDO essentially wrap data sources and fully control access to them via a set of strongly-typed or dynamic interfaces. SDO offer a number of advantages: data encapsulation, better semantics in comparison to the previous approach (because data manipulation is organized around data objects from application domain), better modularity and reuse. However, SDO have a weakness (as in the case of data-as-a-service approach): the problem of interface design is not solved. Therefore, granularity mismatches are possible, but the usage of dynamic interfaces can alleviate the problem at the cost of code complexity.

IV. BUSINESS OBJECT QUERY LANGUAGE

A. Concept

As one can see all approaches have advantages and disadvantages. SQL as a data manipulation API provides great flexibility by allowing to construct queries that match the granularity of any information need. However, SQL exposes too much control over the database and circumscribes business logic rules. The data-as-a-service and SDO approaches, on the other hand, enforce business rules by exposing a set of Web operations, which encapsulate data manipulation and hide data organization. However, the granularity of the exposed operations often does not match the needs of application developers due to poor interface design.

Forming a business object out of semantically related data and accessing the data via a fixed set of operations give good control over the data. On the other hand, exposing the data via a specific interface limits application developers in ways they can manipulate the data. To overcome this deficiency we suggest to use only a dynamic interface of a business object and automatically generate the sequence of calls to its operations, given a special query (compact, formal description of an action a developer wants to perform on the business object).

Despite the diverse semantics of business objects they all have the same structure (an array of attributes and associations) and behavior (a set of operations). The most basic set of operations a business object supports is called CRUD - Create, Retrieve, Update, Delete. Although too generic, the operations have an advantage that any business object can support them. Therefore, all business objects can be derived from the same base class featuring the mentioned arrays (of attributes and associations) and CRUD-operations. Such uniform behavior and structure allow introducing a query language for business objects very much like SQL for relations. An important difference to SQL is that a query is translated into a sequence of CRUD-operations, rather than relational algebra operations. CRUD-operations, in turn, are translated into the calls to storage engine API. Hence, we propose the following scenario:

1) A programmer composes a query, the description of what to retrieve from or change in the system, according to formal grammar, and sends the query as a string to the system by calling a generic Web service operation, for example ExecuteQuery.

2) The system parses the string to detect involved business objects and operations on them and builds a query tree - an internal representation of the query. The tree is then passed for further processing to a query execution runtime.

3) The runtime obtains references to the business objects, on which the operations must be performed. Then the runtime traverses the query tree in a specific order and converts recognized query tokens to appropriate CRUD-calls on the source business objects. For example, tokens from select clause are converted to Retrieve or RetrieveByAssociationChain operations, while tokens from update clause are converted to Update operations.

4) Having constructed the call sequence, the runtime binds corresponding string tokens to the input parameters of CRUD-operations. For example, the token Customer.Name of the select clause is interpreted as a call to Retrieve operation with the input parameter value “Name” on the object Customer. Now everything is ready to perform CRUD-operation calls in the on-the-fly constructed sequence.

5) The last step is the composition of result set. The result is compiled in an XML document and sent back to the calling application.

In its essence BOQL performs on-the-fly orchestration of calls to objects’ CRUD-operations based on user-defined queries. BOQL queries are transformed to a sequence of calls that perform the required action. This transformation is made feasible by a uniform representation of business objects (in terms of the structure and behavior). BOQL has an advantage over the data-as-a-service and SDO in supporting queries at any granularity level (due to the usage of dynamic interface of business objects). At the same time, BOQL does not
circumvent business rules (because CRUD-operations control data manipulation and enforce business logic rules), which favors it over the API based on standard SQL.

In order to implement BOQL the following system elements are required: business object (BO) engine, query engine and storage engine. BO engine is instantiated at the system’s start up time to assemble business objects and store references to them in a special pool. The query engine (to be more precise - the runtime environment) will access the pool during BOQL query handling to select, update, insert or delete data from business object instances. Because business objects are independent from each other they are instantiated in parallel. This greatly improves the system’s start up time. Every business object exposes the mentioned CRUD operations. The operations implement business logic of an object by manipulating raw data via the low level interface of the system’s storage engine. Storage engine is responsible for issuing calls to the operating system in order to access either in-memory or on-disk data files.

To the query engine ERP data are seen through the interfaces of business objects: a collection of attributes and associations to other objects and CRUD operations. The direct access to the data is prohibited to enforce integrity rules and internal business logic implemented by business objects. To access the business data from either the application layer or an external application the later must use the standardized interface exposed by the query engine.

Query engine is also created at the system’s startup time. It is a single point of access into the system for all data queries. Therefore, it is a potential bottle-neck in the system. To make query engine scalable it has to support multi-threading and allow processing queries independently from each other. Many threads of execution inside the query engine substantially increase it’s throughput. When the engine receives a query, it should acquire a thread from the system’s thread pool. The worker thread should do the processing: first, the query is parsed and recognized query tokens are transformed to corresponding operation invocations; second, the work is handed over to the runtime environment that is responsible for performing actual service calls on business object instances and constructing the result set. Whenever is possible, the CRUD-calls are done in parallel. To figure out what calls to execute at the same time the runtime analyzes the query tree built by the parser for a given query and issue every call, which does not rely on yet not retrieved data, in a separate thread. The result of these invocations is assembled in a single XML document and sent back to a working process.

The current work does not provide an in-depth formalism for BOQL (i.e. optimization algorithms, in-memory data layout, etc.). Over-constraining BOQL with such formalism may divert archtects from it. For example, supports of column-orientation will oppose row-oriented organization of main memory storage of business objects. The main contribution is rather a set of architectural and engineering guidelines on turning BOQL into a primer data manipulation API for an ERP system.

**B. BOQL grammar**

The formal grammar of BOQL is almost fully resembles that of data manipulation language (DML) constructs of SQL. As SQL DML, BOQL supports SELECT, UPDATE, DELETE and INSERT statements. It also has SELECT, FROM, WHERE, GROUP BY, ORDER BY and HAVING clauses. Moreover, the standard SQL aggregate functions like COUNT(), AVG(), SUM(), MIN() and MAX() are supported. BOQL has full support for arithmetic expressions, nested expressions, comparisons and predicates (LIKE, BETWEEN, IN, AND, OR and others). The choice of SQL as a reference language was not a coincidence. The main reason is that we wanted BOQL to be as expressive as SQL. With SQL a developer can access any data (at any granularity level) with little effort (minimum number of statements). The second reason was the willingness to minimize the learning and migration effort: for years SQL has been the primer choice of data manipulation in ERP systems. Therefore, having BOQL similar to SQL will somewhat simplify the transition.

Apart from some minor syntactic differences, BOQL differentiates from SQL in three ways, which essentially position it apart from SQL dialects.

1) BOQL natively supports business object hierarchy and enforces business logic. Because of tight integration with BO engine, the type system of the later is fully shared with query engine. This is the biggest advantage over SQL, which has no knowledge of business objects. For SQL this essentially implies, that objects must be reconstructed in the application layer of the system, whereas with BOQL they are embedded into data storage.

2) BOQL does not support relations. Because ERP data is inherently object-oriented, there is no need for supporting plain relations.

3) BOQL query always has one source business object. Because business objects are connected with associations and BOQL “understands” this, there is no need for explicit joins in FROM clause. Joins are implied by associations and, thus, can be omitted. The later fact greatly simplifies BOQL queries in comparison to SQL.

An example of BOQL query is provided in the subsection V-C. Please note the simplicity of the query, despite it results in four join operations (SalesOrder object joins SalesOrderItem and Contact objects, Contact object joins Address and Customer objects).

**C. Reducing BOQL overhead with query compilation**

In order to execute a BOQL query, a system must perform a sequence of input/output (I/O) calls to manipulate in-memory or on-disk data files. The construction of this sequence is started with query parsing, which basically discovers what actions on what data the programmer wants to be performed. Having parsed the query, the system constructs a query plan - a sequence of CRUD operations that perform the desired action. Often, many alternative sequences can represent the same query. Therefore, the system must use optimization.
techniques to select a sequence that is optimal in terms of a given cost model (in the current work, however, this part is not considered). Once the optimal query plan is obtained the system executes it. All queries (and thus query plans) are specified in terms of a business object model, while execution of the query must be performed against a set of physical files. Therefore, the system must map the logical schema to physical files. This is exactly what metadata (or catalog) does for the BOQL: it specifies the structure of business objects (attributes and associations) in terms of physical storage. Thus, having constructed the query plan and resolved objects’ elements into physical locations, the system (namely its storage engine) makes a number of calls to the operating system’s I/O API to manipulate the physical data.

As one can see a lot of decision making is involved in BOQL query execution process. Moreover, the mentioned actions take place every time a query is processed. In many cases this is not practical. If a query occurs periodically, for example as a part of report generation or an OLAP application, then postponing the decision making to run time adds no value and produces overhead. Instead of going through all of the steps every time, the system can perform them just once (during the first call or even at design time for the cases when queries are known before the execution of a program) and cache the decisions in order to reuse them for later calls.

As was presented in subsection IV-A, BOQL is basically an interpreter that transforms a string values into calls on business objects. By turning BOQL into a compiler this transformation can be shifted from run time to compile time and hence performed only once per individual query. In this case the behavior of the system changes to the following:

1) A programmer composes a query and sends the query for compilation to the system.
2) The system parses the string and builds a query tree.
3) The system traverses the query tree and emits code (in the language used to program the BO engine) calling CRUD-operations of business objects via BO engine interface.
4) The code gets compiled and linked with the compiler and linker used to compile and link the BO engine.
5) The resulting binary code is stored inside the BO engine.
6) Whenever the programmer needs to execute the query they simply call the compiled code.

Thus, by pre-compiling a query and storing the native instructions inside the BO engine the overhead of creating CRUD-call sequence can be fully avoided. This by the way can be used in the UI configuration scenario (see subsection V-A): all queries are known at the systems’ design time, and therefore are completed already into corresponding CRUD-call sequence.

A significant performance improvement is achieved by translating an execution plan directly to a sequence of calls to the API of the underlying storage engine. In other words the execution plan is compiled into a program and linked against the system. This fully eliminates metadata look ups (i.e. the resolution of data model into physical files) at run time, meaning that the mapping of logical data schema to its physical representation will be shifted from run time to compile time. Thus, instead of interpreting query plans the system will dynamically load the corresponding programs and execute them.

If compilation improves performance of query execution, why not always compile queries instead of interpreting them? There are a number of reasons for why not. Firstly, compilation is not free. For rare or ad-hoc queries the savings in schema mapping may approximately be equal to the overhead of compiling and dynamic loading. Secondly, because of changes in statistics the optimal query plan may become no longer optimal and the query will not be executed the best way possible. Thirdly, allocation of compiled code reduces the amount of available main-memory. This problem can be partially alleviated by parametrization of compiled query execution plans (parametrization is left for the future work). This will increase the reuse rate of compiled plans: multiple queries differing only in values of supplied variables will be assigned to the same block of compiled code. The reduction of memory consumption can also be achieved by further constraining the set of queries that are allowed to be compiled. For example, not only must a query reoccur, but also it must reoccur at a given rate (e.g. once in two seconds or once in 500 ms.), or the system can allow the query to be compiled only is the later has occurred at least N times. In other words, the system can employ different policies that assure "true" re-occurrence of a query. The next, fourth, problem with compiled queries is their invalidation due to changes in data files. Such changes may result from data schema alterations (e.g. adding and deleting attributes of existing entities or creating and dropping entire entities) or data changes (e.g. insert, update and delete operations on existing entities). In this situations compiled code with already resolved logical schema (i.e. with "hard-coded" data locations) gets desynchronized with the content of corresponding data files. Such code must be recompiled in order to perform desired operations.

Hence, query compilation offers performance improvements, but requires additional resources and in some situations may come at the price of lower flexibility in comparison to query interpretation. Compilation is best suited for SELECT-like queries against large and unchanging data. Such queries are common to OLAP and reporting scenarios.

D. Prototype of BOQL

The current subsection demonstrates a possible implementation of an ERP system built according to the principles discussed earlier. Figure 1 sketches the architecture of a prototyped system. It has five elements: user interface (UI), a user request handing and dispatching layer, a query engine, a business object engine and a storage engine.

The prototype supports two types of UIs: (i) the one connecting directly to the back-end system and profile storage, executing BOQL queries on its own and rendering UI-forms on the client side - fat client and (ii) the one delegating rendering the UI to the request handling and dispatching layer - thin
clients. This layer is responsible for generating UI-forms for a client given the client’s configuration profile (see Section V-A). The layer exposes a Web service based API. Because this Web service can easily become a bottleneck in the system, we factored out the actual configuration lookups and query execution from the Web service to reduce its workload and thus improve the responsiveness of the overall system. For this we run so called working processes. Each process can be run on any physical server, to which the Web service hosting machine can establish a TCP connection. The Web service in its essence is simply a wrapper that provides a standard-based (SOAP/XML) interface to working processes.

The prototype has been developed using LAMP (Linux, Apache, MySql, PHP). The query, business object and storage engines are implemented and deployed as C++ console applications. The SOAP interface of the query engine is published as an Axis2/C Web service. The lexer and parser for BOQL (both interpreter and compiler) were generated with GNU Flex and GNU Bison respectively. For the in-memory data storage two options have been used: Berkeley DB (configured as in-memory storage) and SAP Business Intelligence Accelerator (which can for simplicity be considered as a column-oriented main-memory based storage). The archive database is implemented with MySQL Database Server 5.5. The storage engine and BO engine are deployed on the same dual-processor blade (Quad-core Intel Xeon E5450 3.00GHz with 32 GB of RAM) run by SUSE Linux Enterprise Server 11. The query engine is deployed on a blade with the same software/hardware configuration, except for 1 GB of available RAM. The engines use TCP connections to communicate with each other. The user request handling and dispatching layer is implemented as a C++ console application and deployed on a dual-processor blade (Quad-core Intel Xeon E5450 3.00GHz with 1 GB of RAM) running SUSE Linux Enterprise Server 11.

E. Exposing Business Object Model

To use BOQL an application programmer must know what business objects the system has, and what attributes and associations every business object has. To communicate this information we use oriented graphs. The vertices of a graph denote business objects and oriented edges denote associations. A set of attributes is attached to every vertex (see Figure 3)\(^1\). The graph plays the same role for business objects as the schema for a database.

We have developed a tool called \textit{Schema Explorer} that automatically retrieves metadata from the test system and builds a business object graph. Such a tool greatly simplifies the creation of BOQL queries. This tool provides a plenty of useful functionality: business object search, association and attribute search, finding connections/paths between any two business objects, displaying a business object graph or its part, IntelliSense-like support for query editor, test execution of a query, to name just a few.

F. Sustaining Changes

If a business object graph changes, previously written BOQL queries and all applications issuing these queries may become invalid. The situation is the same as in the case of changing the schema of a database - existing SQL queries are not guaranteed successful execution. Therefore, the suggested API and its implementation must be augmented with a compatibility assurance tool.

A literature review showed that in the case of relational and object-oriented databases the view mechanism can be applied to cope with schema changes \cite{7}, \cite{8}. The prime tool for view support in relational and object-oriented databases is mapping \cite{9}, \cite{10}, \cite{11}. BOQL natively supports mappings. That is, mappings can be seamlessly embedded into the system. The mappings can be supported in two ways: by means of data access plug-ins and query rewriting.

The first approach is based on substitution of CRUD-operations at run-time. This is achieved by encapsulating a new version of the operation inside a function and dynamically selecting this version when handling a request. Every time a CRUD-operation is called, the system detects which implementation must be used. The choice depends upon the version identifier that a client application sends with its request.

\(^1\)For the sake of compactness we do not list the attributes on the diagram.
request. The substitute version is implemented as a function of a feature pack that is dynamically loaded by the system when it encounters a query addressed to the older version of the schema for the first time. Having loaded the feature pack, the system registers the loaded CRUD-operation with the query engine. This approach assumes that the system’s provider must develop and deploy corresponding feature packs.

The second approach is based on altering a query while it is being parsed. Before converting a query token to an appropriate operation call the parser looks up a correspondence dictionary to find the actual path to the asked attribute or association, rewrites the token and re-parses it to get a valid CRUD-operation call sequence.

V. APPLYING BOQL

A. Profile-based configuration

The idea of profile-based user configuration aims at enabling end-users of a system to customize the system’s presentation layer according to their own preferences. This is achieved as follows. With the help of Schema Explorer end-users discover information in an ERP system that they are interested in. They simply select business objects of interest and select the attributes they want to see for every object. Then with the help of the same tool they generate BOQL that retrieve/change these data and store these queries in a structured way in a personal profile. Next, when a user logs in to the system the later picks up the user’s profile executes necessary queries and presents the results to the user.

The UI of the prototype is capable of automatically generating three different types of front-end forms: (i) business object summary form: lists all instances of a certain type with a short description for each instance; (ii) drill-down form: lists detailed information on a particular business object instance; (iii) related items form: lists instances of other business objects that are connected with a given object instance. Figure 2 illustrates two forms: the upper one is a summary of all customers in the system, and the lower one is a drill-down form for a particular instance of a sales order business object. The related items form is essentially the same as the summary form. There is no difference in rendering them. The only difference is the actual BOQL query, the result of which populates the form. On the right side of the figure there is a profile from which the forms were automatically generated. One can see that a user called “Purchase01” is interested in business objects Customer, Opportunity, Material, Sales order, etc. (all second level elements of the tree). Within each object there is a list of fields that must be displayed for the object. For “Customer” the fields are name, status and id. An object instance can also have related items associated to it. For example, when the user looks up an instance of a sales order they might be willing to navigate to services, quotes and opportunities associated to this particular sales order. Note that the profile is fully configurable and no query is executed before the user has explicitly clicked a corresponding link.

To save time and effort of end users an ERP system vendor can create a number of role-specific configuration profiles, which users may adjust according to their needs.

B. Navigation in ERP Systems

Data in an ERP system are stored in a normalized form. Therefore, it is almost for certain that semantically related data pieces will be disjoint by the system. Even though business objects aim at creating a semantically complete entities, they do not reassemble business data completely after the normalization at the storage layer. This results in a partial loss of semantical links between data in the ERP system. The reason is that many semantical relationships are not modeled in the database. For example in SAP R/3 system, there is no direct physical relationship between a customer and invoice entities. They are connected via a sales order. Therefore, if a user wants to know if a customer paid their invoices the user needs to find first all sales orders of the customer and then check all associated invoices. This is not practical.

The field research, which we conducted, showed that users of ERP systems are struggling a lot with this problem. They often find that even though their system stores all information they need, they cannot get it quickly because of the missing links. We have observed that a user opens on average 3 to 6 windows, performs 10 to 20 mouse clicks and types 15 to 30 characters to get semantically related data. Thus, searching and looking up data in the ERP system take considerable amount of an employee’s working time.

The ability to navigate easily in an ERP system greatly improves the productivity of the system’s users. The opposite is also true. If an employee needs to spend considerable amount of time to look up transactional or master data in the ERP system, the employee’s productivity significantly drops.
BOQL in combination with the Schema Explorer and user profiles can solve this problem completely.

To reconstruct semantical links between business objects we run a path search algorithm on the business object graph. In the example above the user would need to select in Schema Explorer the Customer business object as the starting point and CustomerInvoice as the target business object. Then the tool would use a modification of a graph traversal algorithm to find a path from the source to the target and convert this path to a BOQL query. After this the user just needs to update their profile (the section Related Items of the Customer business object). Next time when the user opens customer details screen a corresponding link will appear (see Figure 2).

C. Enterprise Composite Applications

Consider a Web retailer that sells items on-line and subcontract a logistics provider to ship sold products. The retailer operates in a geographically large market (e.g. the US or Europe). In this situation the consolidated shipment of items can generate considerable savings in delivery. Consolidation means that a number of items is grouped in a single bulk and sent as one shipment. The savings come from price discounts gained from higher transportation volume and less transaction cost per item shipped (because it is the bulk that is charged, but not individual items).

To manage their sales, the retailer uses a system with the business object graph as Figure 3 presents. We assume that the system exposes a query-like Web service interface as described in the section IV-D. The query returning the shipping address for all sales order items that are to be delivered looks as described in the section IV-D. The query returning the shipping address for all sales order items that are to be delivered looks as follows:

```
SELECT
    SO.id, SO.Contact.Address.post,
    SO.Items.id, SO.Contact.Customer.name
FROM
    SalesOrder AS SO
WHERE
    SO.status = "ToDeliver"
GROUP BY
    SO.Contact.Address.city
```

By invoking the query-like Web service and passing the above query to it, a third-party application consolidates the items by their destination. The next step for the application is to submit a request for quote to a logistics provider and get the price of transporting each group of items. Many logistics providers have a dedicated service interface for this, so the application can complete this step automatically. Once the quote has been obtained and the price is appropriate the products can be packaged and picked up by the logistics provider.

VI. CONCLUSION AND FUTURE WORK

Efficient data manipulation APIs are essential for ERP systems. They help to resolve a number of challenges. Unfortunately, existing approaches and APIs do not offer an appropriate level of flexibility and simplicity while guaranteeing integrity and consistency when accessing and manipulating ERP data.

This paper contributes the concept of query-like service invocation implemented in the form of a business object query language (BOQL). BOQL is the corner stone of an API offering both the flexibility of SQL and data encapsulation of SOA. In its essence, BOQL is on-the-fly orchestration of CRUD-operations exposed by business objects of ERP systems. Furthermore, the paper showed how BOQL enables navigation among ERP data and configuration of the UI layer as well as development of enterprise composite applications. The paper also outlined the major components of the architecture of BOQL and a prototype ERP system that supports BOQL as its primary data manipulation API. In order to improve the performance of BOQL an option of compiling BOQL queries was considered and prototyped.

The future research will be dedicated to the in-depth BOQL optimization. In particular, the applicability of conventional database management systems query optimization approaches will be evaluated.

REFERENCES