FLEXIBLE DATA ACCESS IN ERP SYSTEMS

Vadym Borovskiy[1], Wolfgang Koch[2] and Alexander Zeier[1]
[1]*Hasso-Plattner-Institute, Potsdam, Germany*
[2]*SAP AG, Walldorf, Germany*

Keywords: ERP data access, Business object query language, Navigation in ERP systems, Enterprise composite applications, UI customization.

Abstract: Flexible data access is a necessary prerequisite to satisfy a number of acute needs of ERP system users and application developers. However, currently available ERP systems do not provide the ability to access and manipulate ERP data at any granularity level. This paper contributes with the concept of query-like service invocation implemented in the form of a business object query language (BOQL). Essentially, BOQL provides on-the-fly orchestration of CRUD-operations of business objects in an ERP system and allows to achieve both the flexibility of SQL and encapsulation of SOA. To demonstrate the power of the suggested concept navigation, configuration and composite application development scenarios are presented in the paper. All suggestions have been prototyped with Microsoft .Net platform.

1 INTRODUCTION

Flexible data access is essential for Enterprise Resource Planning (ERP) systems. Efficient retrieval and manipulation of data are required to address a number of acute needs of ERP systems users and application developers. By interviewing users of ERP systems we found out that two the most common requirements of every-day system users are: *(i)* efficient *navigation* among ERP data and *(ii)* user-specific *configuration*. The first one means dynamic and fully automatic discovery of information semantically relevant to a given user in a given context. For example, when a sales manager views customer details the system provides a list of links to invoices not paid by the customer or a list of links to products ordered over the last six months by this customer. By clicking the provided links (basically shortcuts) the manager can navigate to relevant data with minimum effort. The second requirement means that a pre-packaged system must support user-specific views on top of standard data structures. For example, a sales order in SAP R/3 has hundreds of fields, most of which are never used. Despite this all fields are displayed at the sales order entry form. This complicates the form and slows down order processing.

To satisfy these requirements an ERP system must support data retrieval and manipulation at any granularity level. Indeed, if an ERP system had application programming interface (API) allowing to query any piece of data and assemble dispersed data piece in a single result set, the two requirements would be feasible to fulfill. In fact, a solution to the second one would become trivial. A personal configuration would be nothing else but a set of queries returning/processing only relevant attributes. The navigation challenge could also be resolved in the same way: the list of links could be constructed based on the result set of queries. More details on that can be found in the Section 3.

In addition to the user needs flexible data access is relevant for ERP application developers in a number of use cases. A series of interviews conducted by the authors revealed the need for an API as discuss above for integration and extension scenarios. In the first case, the lack of a convenient API prevents application developers from exchanging data between an ERP system and other software used by an enterprise. In the second case, current poor APIs significantly complicate the development of enterprise composite applications (ECAs) on top of ERP systems. Enterprise composite applications have become a primary tool of extension and adaptation of enterprise software. They allow application developers to add features to ERP systems and, thus, view an ERP system not as a product but as a platform exposing data and functionality, which can be reused/recombined in new ways. Turning an ERP system into such a plat-

Borovskiy V., Koch W. and Zeier A. (2010).
FLEXIBLE DATA ACCESS IN ERP SYSTEMS.
In *Proceedings of the 12th International Conference on Enterprise Information Systems - Databases and Information Systems Integration*, pages 96-104
DOI: 10.5220/0002976700960104

form inevitably requires a quite high degree of system openness and, thus, appropriate API.

The current work contributes with the concept of query-like service invocation implemented in the form of a business object query language (BOQL). BOQL is the corner stone of the API offering both the flexibility of SQL and encapsulation of SOA. Section 2 of the paper presents this concept along with a prototype demonstrating the idea in practice. Section 3 demonstrates the application of BOQL to the challenges and needs discussed above. Section 4 concludes the paper.

2 FLEXIBLE DATA ACCESS

Data access API of an ERP system highly depends on internal details of the system. Not all architectures can efficiently support data access from outside of the system. In fact, most of the systems never allow such access. Data accessibility means an ability to access the data model (or metadata) and actual data storage. In other words, users need to know how their business data is structured and how they can retrieve the data.

2.1 State of the Art

A straightforward approach to data access can be to use SQL. Since ERP systems rely on relational databases, issue SQL statements could be issued directly against the databases to retrieve required data. Although SQL is natively supported by the underlying databases, this approach is unlikely to deliver the expected results. SQL statements need to be written against an actual schema of a database. The schema of an ERP system is very complex and is not intended to be directly used by customers. In fact, it is considered to be private and therefore is hidden from users. ERP vendors can use the concept of views on top of the internal database. On one hand views can hide data organization and internals. On the other hand views can reduce the essential complexity of the schema from a customer's perspective by exposing only a subset of the schema relevant to a given customer or a group of customers. So why not to use SQL against views as a data accessing API?

The problem with this approach is that it violates the data encapsulation principle. Basically SQL against views exposes too much of control over the underlying database and greatly increases the risk of corrupting data in the system. An ERP system is not only a collection of structured 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 access to the database circumvents the rules and implies data integrity violation. Therefore, to enforce the rules the direct access to data by any means is strictly prohibited. To enforce business rules and further increase the encapsulation semantically related pieces of data are grouped together with business logic (expressed in a programming language) to form a monolithic construct called business object. This allows to hide actual data storage behind an object-oriented layer. Grouping data and business logic in business objects simplifies the consumption of data from a programming language. For these reasons ERP systems can be seen as object-oriented databases. In this case SQL against views as data access API is inappropriate.

An alternative to SQL against views can be data as a service approach. In this case a system exposes a number of Web services with strictly defined semantics. By calling operations of these services required data is retrieved. For example by calling *ReadSalesOrder(soId)* operation of *SalesOrderManagement* service full information on a sales order given its ID can be retrieved. This approach has an advantage of hiding internal organization of data. Instead of a data schema a set of operations that return data are exposed by a system. By choosing operations and calling them in an appropriate sequence required data can be retrieved. 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 Business Suite. Amazon Electronic Commerce service is another example of such approach. However, this method has two serious disadvantages: lack of flexibility and high cost of change. Although an ERP system vendor can define many data accessing operations, they will never cover all possible combinations of data pieces of an ERP system. Often these operations are limited to one business object. ECAs on the other hand address very specific or fine-granular needs and deliver value by assembling information coming from different locations of a system. Therefore, the granularity of data services does not match the granularity of ECAs' operations and the services cannot provide adequate support for the ECAs. For this reason ECAs need to issue multiple service calls and combine a result set on their own. This greatly complicates the development of ECAs and undermines their performance. This situation clearly demonstrates the advantage of the SQL against views approach. The ability to construct fine-granular queries that fully match the information needs of ECAs makes SQL a much more flexible API than data as a service.

High cost of change has to do with evolution. Over its lifecycle an ERP system will go through a number of changes. If these changes affect internal data organization most probably all data services that work with affected data structures will need revision. In the worst case a subset of data service operations may become irrelevant and require full substitution. Revising these operations is costly. The situation exacerbates if changes to service operations make their new versions incompatible to previous ones. This implies failures in ECAs that already consume previous versions. The SQL against views approach has less cost of change. A set of views that map the old schema to the new one localizes changes on the database level and does not require the revision of all outstanding ECAs.

As one can see both approaches have advantages and disadvantages. SQL as a data access API gives great flexibility by allowing to construct queries that match the granularity of a user's information needs. However, SQL exposes too much control over the database, circumvents business logic rules and binds ECAs to a specific implementation platform. The data as a service approach on the other hand enforces business rules by exposing a set of Web operations which encapsulate data access and hide data organization. However, the granularity of the exposed operations does not match that of a user's needs, which creates inflexibility and hits performance. Furthermore, data as a service approach has high cost of changes.

2.2 Business Object Query Language

In this subsection we contribute with an idea of how to combine the advantages of discussed approaches while eliminating their disadvantages and propose a concept called business object query language (BOQL).

It is clear that accessing raw data directly and circumventing business logic contradicts with data encapsulation. For this reason business objects appeared. They fully control the access to the data and protect the integrity of data. From external perspective business objects are simply a collection of semantically related data, e.g. *invoice, bill of material, purchase order*, and a number of business operations that can be performed on the data, e.g. *read, create, etc*. A business object can be represented as set of data fields or attributes, e.g. *id, count, name*, and associations or links with other business objects, e.g. a *SalesOrder* is associated with a *Customer* and *Product* business objects.

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 known as CRUD - *Create, Retrieve, Update, Delete*. Although too generic, this set of operations has an advantage that any business object can support it. 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 to introduce a query language for business objects very much like SQL for relational entities. We propose the following scenario:

1. A programmer composes a query, the description of what to retrieve from the system, according to some SQL-like grammar and sends the query as a string to the system via a generic service operation, for example *ExecuteQuery*.
2. The system parses the string to detect present clauses (*from, select, where,* etc.) and builds a query tree - an internal representation of the query. The tree is then passed for further processing to a query execution runtime, very much like in a DBMS).
3. Using the *from* clause the runtime obtains references to the business objects from which the retrieval must be performed: source business objects. Then the runtime traverses the query tree in a specific order and converts recognized query tokens to appropriate operation 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 business object Customer. Now everything is ready to perform the calls of CRUD-operation in the on-the-fly constructed sequence. The last step the runtime performs is the composition of result set. After that the result is formatted in XML and sent back to the calling programm.

In its essence the query language performs orchestration of calls to objects' operations based on user-defined queries. These queries are transformed to a sequence of operation calls that yield the required data. Business object query language has an advantage of supporting fine-grained queries as in the case of SQL without circumventing business rules as in the case of the data as a service approach. Such an

approach is allowed by a uniform representation of business objects (in terms of the structure and behavior).

An important question to answer is what grammar the business object query language must have. We think SQL- or XQuery-like grammar is the most appropriate choice. The former one assumes business data to be organized as a set of relations. The later one sees business data organized in documents. Both notations are of equal power and allow for querying at a business object level (meaning the attributes of a single object) and at a cross business object level (meaning business object joins). The semantics of both suggested grammars is straightforward to understand by developers. In fact, many of them may have already worked with either one, and hence will quickly learn the business object query language. An ERP provider can support both (SQL- and XQuery-like) notations. In the subsection 2.3 we will briefly mention what is needed for this.

Although the suggested approach offers great extend of data access flexibility, it creates a number of challenges. The first one is the development of the query parser. If an ERP provider decides to support most of the SQL features/syntax, the implementation of the parser (and runtime) will require high effort. Therefore, there is no sense in pursuing the suggested approach for systems with simple data schema. For large systems like SAP Business ByDesign, where there are hundreds of business objects with dozens of associations and attributes each, this approach is more preferable. The parser and the runtime are independent from business objects. Once the both have been developed a provider can use them without changes to expose new objects.

Our second concern regarding business object query language is performance. Users may create very complex queries and thus put a high workload on the system. Given the fact that the system is planned to be shared by many customers, a high workload generated by one customer application can potentially affect the performance of other customers' applications. What is more disturbing is the fact that arbitrary queries may destroy the performance of the underlaying database. Every database management system has internal query optimizer and a set of system tables accumulating the operational statistics. Given the statistics and database metadata (primarily the information on indices and the size of tables), the optimizer computes the query execution plan that ensures the highest possible performance for a given query in a given situation. The problem here is that the optimizer adjusts itself to the most frequent types of queries. But from time to time it will encounter a query for which it will

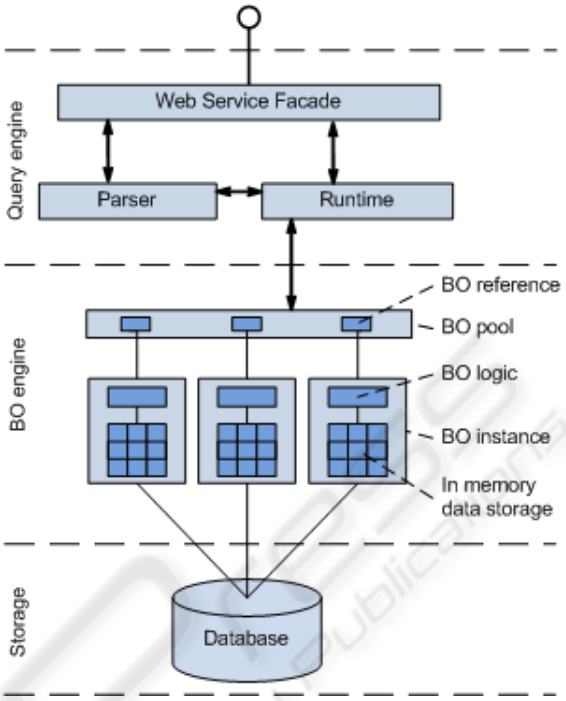

Figure 1: The architecture of the test system.

generate the plan much worse than its default plan, meaning that optimization will only worsen the performance. Such situations happen periodically with many databases. The reason for this is that the internal statistics has been computed for completely different types of queries and the system has sharpened itself for those queries. In such situations the optimizer can generate a weird plan (e.g. use wrong index) and the query will block the system for some time. In the worst case such rare queries may trigger the recalculation of the statistical information and readjustment of the optimizer, which blacks the whole system for much longer time.

In our implementation we have not experienced such performance problems for the reason we have cached the data in in-memory tables and did not issue on the fly any queries to the underlaying database. In fact, using in-memory data storage solves the performance problem mentioned above. Nevertheless, we believe that such a problem is worth an additional research and investigation.

2.3 Suggested Architecture

The current subsection demonstrates how an ERP system can support BOQL. The Figure 1 sketches the architecture of a prototyped system. BOQL is implemented by two elements: a business object engine and a query engine: the former manages business ob-

jects in a way BOQL assumes[1] and the latter provides access to them from outside of the owning process via a query-like interface. These two elements are instances of *BoEngine* and *QueryEngine* classes respectively. Both are created at the system's startup time. Business object engine is instantiated first to assemble business objects and store references to them in a pool. Then the instance of the query engine is created. It has access to the pool and thus can manipulate the objects.

Every business object encapsulates an in-memory table to cache data. The in-memory table is populated with data taken from a private database. Every object also encapsulates logic to synchronize its in-memory table with the database. To the query execution runtime an object is seen through its interface: a collection of attributes and associations to other objects and a set of operations. How those are implemented is completely hidden inside the object. Typically, attributes and associations are bound to data fields and relations of the underlying physical storage. In this prototype we concentrate on two operations (accessors) from the interface: *Retrieve* to get attributes of a given object and *RetrieveByAssociationChain* to navigate from one object to another via specified associations. These operations retrieve data from the underlying physical storage or the local cache according to internal business logic. By default the accessors assume one-to-one correspondence between a business object's logical and physical schemas. For example, if an attribute *Attr1* of a business object *Bo1* is queried the query runtime looks for data field named *Attr1* in a table corresponding to a given business object; if an association *Assoc1* of a *Bo1* is queried the runtime looks for a foreign key relationship corresponding to the association and constructs a join.

Neither the business object nor its in-memory and database tables can be directly accessed outside of the owning process. 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 an external application must use the standardized query-like interface exposed by the query engine. When the latter receives a query it parses it and transforms recognized tokens to corresponding operation invocations. The result of these invocations is put in XML format and sent back to a client application.

As an implementation platform for the prototype we chose .NET. The system is implemented as a Windows service and the query interface is published as a Web service hosted by Internet Microsoft Information Services (IIS). The Web service is meant to dispatch a query to the system and serves as a request entry point. There is no other way to invoke or access the system except for issuing a call to the Web service. The physical data storage is implemented as a Microsoft SQL Server 2005 database.

[1]meaning that it guarantees the compliance to CRUD-interface

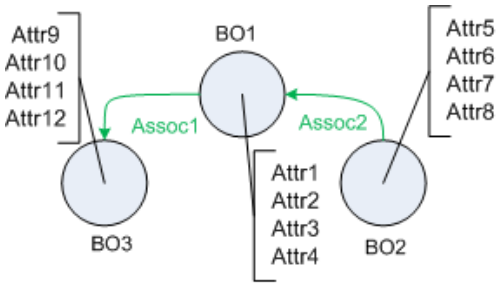

Figure 2: Business Object Graph.

2.4 Exposing Business Object Model

Using BOQL requires the knowledge of business object model, that is what business objects a system has, 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 2). For the sake of compactness we will not list the attributes on further diagrams. The graph plays the same role for business objects as the schema for a database. It depicts the structure of business data and is essential to know to compose queries.

We have developed a tool called *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 support for query editor, test execution of a query, to name just a few. The implementation of metadata retrieval depends on the implementation of the backend system. For the implementation presented in the subsection 2.3 the metadata about the instances of business objects is obtained using the reflection mechanism of .Net Framework. Query engine exposes a number operations which internally use reflection in order to query the business object metadata. For example, if a developer wants to know what business objects a system has the tool calls an operation which scans the pool and obtains the types of business objects instantiated by the system. To look up the list of associations of a given business object,

say Customer, the tool issues a call to another operation that gets the names of elements in the *Associations* array of the corresponding business object. Because business object model does not change (at least should not) we cache metadata in order to avoid the reflection overhead.

2.5 Sustaining Changes

The flexibility of BOQL in supporting fine-grained queries comes from the way it is designed, but the ability to sustain changes is not straightforward to see. In fact, if the business object graph changes, previously written queries and thus 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 architecture must be augmented with a compatibility assurance tool that minimizes incompatibility in the case of changing the business object graph.

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 (Curino et al., 2008), (Banerjee et al., 1987), (Bratsberg, 1992). The prime tool for view support in relational and object-oriented databases is mapping (Bertino, 1992), (Shiling and Sweeney, 1989), (Liu et al., 1994), (Monk and Sommerville, 1993). The architecture we proposed 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 default association and attribute accessors (*Retrieve* and *RetrieveByAssociationChain*) with custom ones. This is achieved by encapsulating a new accessor inside a call-back operation and dynamically selecting this operation when handling a data request. Associations and attributes that have or require custom accessors we call virtual. The set of all virtual elements of a business object is called the view of the business object. Every time a virtual element is called a custom accessor implementing the mapping is invoked. This accessor is implemented as a callback operation 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 plugs in the custom accessor into the runtime of the query engine. In the prototyped system we used reflection mechanism to redirect calls from default to custom accessors.

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 operation call.

3 APPLYING BOQL

3.1 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.

In our prototype we have implemented user interface layer with Microsoft Silverlight. The UI is capable of automatically generating three different types of frontend forms: (i) business object summary form: lists all instances of a certain type with a short description for each instance; (ii) drilldown 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 3 illustrates two forms: the upper one is a summary of all customers in the system, and the lower one is a drilldown 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

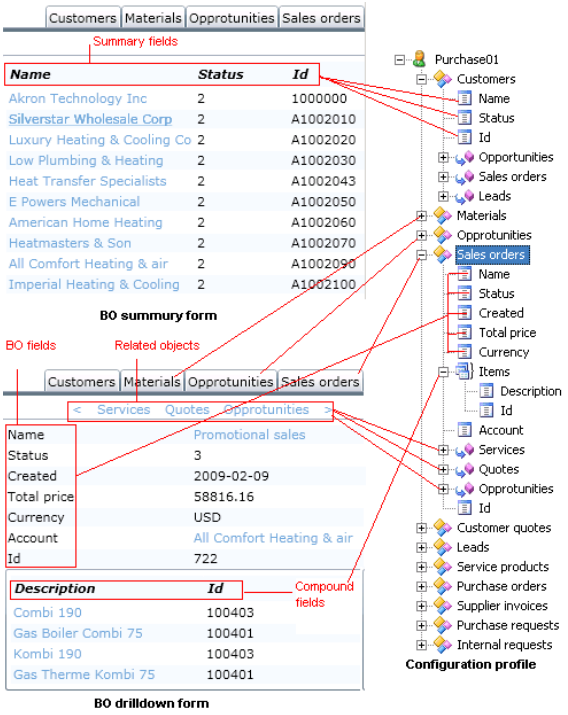

Figure 3: User interface forms and profile configuration.

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.

3.2 Navigation in ERP Systems

An ERP system stores all facts and figures about a company's business activities in a structured way. Conventionally the data are stored in a normalized way. This means that data generated by the same business process is very likely to be split into pieces, which will be stored separately. 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 data is normalized at the storage level of the system. The bigger the company the more diverse and complex data structures get and the more complicated the reassembly becomes. 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 at the database level. 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 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. This results that an employee spends considerable amount of their working time on searching and looking up data in an ERP system.

The ability to handle the corporate data efficiently greatly improves the productivity of the company's employees. The opposite is also true. If an employee needs to spend considerable amount of time to look up transactional or master data in an ERP system, the employee's productivity significantly drops. Unfortunately, this is often the case. 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 3).

3.3 Enterprise Composite Applications

A composite application (CA) is an application generated by combining content, presentation, or application functionality from a number of sources. CAs aim at combining these sources to create new useful applications or services (Yu et al., 2008). An enterprise composite application is a CA application which has an ERP system as one of its sources[2]. CA access their sources via thoroughly specified application programming interface (API). The key characteristics of a CA are its limited/narrowed scope and straightfor-

[2]From now on we consider only enterprise composite applications. The terms "CA" and "ECA" for the sake of brevity are considered to mean the same in the rest of the paper.

ward result set. CAs often address situational needs and provide replies to fine-grained information requests. They are not intended to provide complex solutions for general problems rather they offer compact answers to clear-cut questions.

CAs create value by pulling all data and services a user needs to perform a task on a single screen. These data and services can potentially come from many sources, including an ERP system. Very often users are confronted with a problem of having necessary information and functionality distributed across many forms. By creating a CA that assembles them on the same screen users can substantially increase the productivity of their work. Additional benefit here is that a CA can present information in a way that meets personal preferences of a user.

The architecture we suggest natively supports CA. The main enabler of composite applications is the BOQL, which basically provides a mechanism for query-like invocation of business objects' services. BOQL allows CAs to manipulate ERP data from outside of a system without violating internal business logic. Because the query engine supports SOAP protocol, CAs can be developed and executed on any platform that is suitable for a user and has support for XML.

The process of developing CAs we see as follows.

1. A user[3] figure out on which business objects they want to perform custom operations. This depends on the actual task and application domain. Then the user composes BOQL queries that will return the data from the business objects. To compose the queries the user can use Object Explorer and Schema Explorer tools described earlier.
2. Using SOAP interface of the query engine the user executes the queries and retrieves ERP data.
3. Using selected programming language the user develops code that operates on the selected data and performs the required operations.
4. In case the CA needs to change the data in the source ERP system it composes BOQL queries that do so and executes the queries using the same SOAP interface of the query engine.

In the following use case the flexible data access of the suggested approach creates business value. Consider a Web retailer that sells items on-line and subcontracts a logistics provider to ship sold products to customers. The retailer operates in a geographically large market (e.g. the US or Europe[4]). In this situation the consolidated shipment of items can generate considerable savings in delivery and thus increase the profit of the retailer. Consolidation means that a number of sold items is grouped in a single bulk and sent as one shipment. The bigger the shipment size, the higher the bargaining power of the retailer when negotiating the shipment with a logistics provider. The savings come from price discounts gained from higher transportation volume[5]. In this way the retailer can lower the transportation cost per sold product. The consolidation of shipment is done anyway by all logistics providers in order to minimize their operating costs[6] By controlling the delivery of sold items explicitly, the retailer captures the savings that otherwise go to a logistics provider.

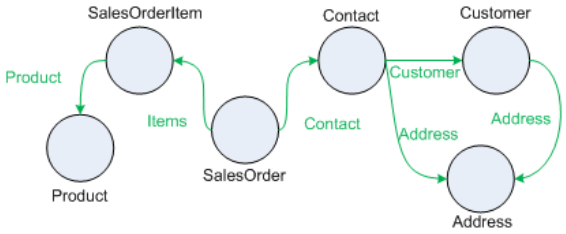

Figure 4: Schema of the Web retailer's CRM.

Let the retailer use a system with a business object graph as Figure 4 presents to manage their sales. We assume that the system exposes a query-like Web service interface as described in the section 2.3. The query returning the shipping address for all sales order items that are to be delivered looks as follows:

```
SELECT
  SO.Items˜id, SO˜id,
  SO.Contact.Customer˜name,
  SO.Contact.Address˜street,
  SO.Contact.Address˜number,
  SO.Contact.Address˜zip,
  SO.Contact.Address˜city,
  SO.Contact.Address˜state,
  SO.Contact.Address˜country
FROM
  SalesOrder As SO
WHERE
  SO.Status = "ToDeliver"
GROUP BY
  SO.Contact.Address˜city,
  SO.Contact.Address˜state
```

By invoking the query-liked Web service and passing the above query to it, a third-party application consolidates the items by their destination. The

[3] power user or application programmer

[4] The greater the territory and the higher the sales volume, the more relevant the case.

[5] So called economies of scale in transportation

[6] In fact, the economies of scale in transportation resulted in Hub-and-Spoke topology of transportation networks.

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.

To enable applications like the one just described the system must expose a flexible data access API. That is, the system must be able to return any piece of data it stores and construct the result set in a user-defined way. As mentioned in the section 2 traditional APIs cannot completely fulfill this requirement: SQL against views circumvents the business rules enforced outside the database; Web services limit the retrievable data to a fixed, predefined set. The architecture suggested in the current work overcomes the existing limitations and offer the necessary degree of data access flexibility.

4 CONCLUSIONS

Flexible data access API is essential for ERP systems. It helps to resolve a number of challenges. Unfortunately, existing approaches and APIs do not offer appropriate level of flexibility and simplicity while guaranteeing integrity and consistency of data when accessing and manipulating ERP data.

The current work contributes with the concept of query-like service invocation implemented in the form of a business object query language (BOQL). BOQL is the corner stone of the API offering both the flexibility of SQL and encapsulation of SOA. In its essence, BOQL is on-the-fly orchestration of CRUD-operations exposed by business objects of ERP systems. In addition the paper showed how BOQL enables navigation among ERP data and configuration of UI layer as well as development of enterprise composite applications. Furthermore, we outlined the major components of the architecture and prototyped with Microsoft .NET platform an ERP system that supports BOQL as prime data access API.

REFERENCES

Banerjee, J., Kim, W., Kim, H.-J., and Korth, H. F. (1987). Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 311 – 322.

Bertino, E. (1992). A view mechanism for object-oriented databases. In *Proceedings of the 3rd International Conference on Extending Database Technology*, pages 136 – 151.

Bratsberg, S. E. (1992). Unified class evolution by object-oriented views. In *Proceedings of the 11th International Conference on the Entity-Relationship Approach*, pages 423 – 439.

Curino, C. A., Moon, H. J., and Zaniolo, C. (2008). Graceful database schema evolution: the prism workbench. In *Proceedings of the VLDB Endowment*, pages 761–772.

Liu, C.-T., Chrysanthis, P. K., and Chang, S.-K. (1994). Database schema evolution through the specification and maintenance of changes on entities and relationships. In *Proceedings of the13th International Conference on the Entity-Relationship Approach*, pages 132 – 151.

Monk, S. and Sommerville, I. (1993). Schema evolution in oodbs using class versioning. In *ACM SIGMOD*, pages 16 – 22.

Shiling, J. J. and Sweeney, P. F. (1989). Three steps to views: extending the object-oriented paradigm. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 353 – 361.

Yu, J., Benatallah, B., Casati, F., and Daniel, F. (2008). Understanding mashup development. *IEEE Internet Computing*, pages 44–52.