SEMANTICALLY RICH API FOR IN-DATABASE DATA MANIPULATION IN MAIN-MEMORY ERP SYSTEMS

Vadym Borovskiy, Christian Schwarz, Alexander Zeier
Hasso-Plattner-Institute, Potsdam, Germany

Wolfgang Koch
SAP AG, Walldorf, Germany

Keywords: Business object query language, Business object virtualization, ERP data manipulation API, Main-memory ERP systems.

Abstract: Assuming the feasibility of main-memory database management systems, the current research aims at designing a new type of data manipulation API, called Business Object Query Language (BOQL), specifically tailored for in-database data manipulation in main-memory ERP systems. The paper contributes the concept of business object virtualization and describes a query processor that takes advantage of this concept. The first serves as a means of grouping raw memory-resident data into high-level data structures, while the second exposes a flexible query-like API to manipulate the high-level data structures. Special effort has been dedicated to integrating the API into C++ programming language.

1 INTRODUCTION

Enterprise resource planning (ERP) systems fully depend on database management systems (DBMSs) in data manipulation tasks. Such a dependency makes ERP systems very sensitive to changes in DBMSs. Because the popular DBMSs were architectured more than 25 years ago for much different hardware than today, their capabilities are far below the level that is achievable with the state of the art hardware (Stonebraker et al., 2007). In particular, the availability of very large and relatively inexpensive memory made possible keeping enterprise-scale data volumes in main memory. Motivated by this opportunity, a number of researches proposed a new design of a DBMS optimized for permanent main memory data residence (Manegold et al., 2000), (Kallman et al., 2008), (Legler et al., 2006). Experiments conducted on a number of different prototypes illustrating the new design showed its remarkable performance advantage (up to two orders of magnitude) over the conventional DBMSs on benchmark workload.

This fact opens a new horizon in ERP systems development: having orders of magnitude performance advantage for in-database operations allows shifting part of business logic from an application server down to the database layer. Getting business logic executed directly inside the database process eliminates expensive data transfers between the application server and the database, removes the need for application-server-side cache and allows performing operations other than declarative SQL data manipulation.

Unfortunately, deploying business logic code inside the database is not that easy because of the mismatch in semantics of the application and the database's data models. That is, business logic accesses ERP data via a semantically rich object-oriented application programming interface (API), whereas a DBMS is capable of exposing only plain relations via semantic-free (generic) SQL. Resolving this mismatch is one of the major milestones on the way towards shifting business logic from application servers to the database.

The main goal of the current research is to leverage the fact of having all database objects permanently living in main memory in order to provide more efficient data manipulation API that ERP systems can employ while executing in-database business logic operations. In particular, in this paper we present the design and prototype of a semantically rich API for in-database data manipulation for ERP systems. The following three points constitute the contribution of the paper:

- virtualization of business objects - the way of resolving high-level business object data model into the primitives of the storage engine;

Borovskiy V., Schwarz C., Zeier A. and Koch W..
SEMANTICALLY RICH API FOR IN-DATABASE DATA MANIPULATION IN MAIN-MEMORY ERP SYSTEMS.
DOI: 10.5220/0003503502530260
In *Proceedings of the 13th International Conference on Enterprise Information Systems* (ICEIS-2011), pages 253-260
ISBN: 978-989-8425-53-9

- business object query language (BOQL);
- integration of BOQL into an object-oriented programming language.

A significant part of the work has been dedicated to demonstrating the feasibility of BOQL, which provides a basis for a larger and more careful effort, including the in-depth development and analysis of formalism for BOQL (i.e. optimization algorithms).

2 RELATED WORK

In this section we briefly review the most important APIs that to our knowledge are used in different state-of-the-art ERP systems.

A straightforward approach to data manipulation is 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 an application 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. 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 (Bernstein et al., 1999). 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 latter 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 (Bernstein et al., 1999). 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 DBMS, which hinders the performance.

An alternative to SQL is data as a service approach. In this case a system exposes a number of Web services with strongly-typed interfaces operating on data. By calling these services application developers perform required data manipulation. 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. An example of such a case is presented in (Grund et al., 2008). To partially overcome the mismatch, the interfaces of Web services can be relaxed (Borovskiy et al., 2009). This, however, will blur the semantics of the operations.

Service data objects (SDO) 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 (Resende, 2007). 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.

3 BUSINESS OBJECT QUERY LANGUAGE

As one can see the APIs used in the state-of-the-art ERP systems have advantages and disadvantages. SQL provides great flexibility by allowing constructing queries that match the granularity of any information need. At the same time, SQL exposes too much control over the database and circumvents business logic rules. In contrast to SQL, object-centric and service-centric APIs enforce business rules and data integrity by exposing a set of operations with well-defined semantics (i.e. strongly-typed interface), which encapsulate data manipulation and hide data organization. However, the granularity of the exposed operations (in terms of record selection and attribute projection) often does not match the needs of application developers.

The main contribution of the current research is an attempt of combining the advantages of the state-of-the-art approaches by leveraging main-memory database technology. The designed API follows data encapsulation principles without restricting the flexibility of data manipulation. Data encapsulation is achieved by organizing data manipulation around business objects. In order to avoid the disadvantages of object-centric APIs (overhead of reconstructing business objects out of raw data and inflexibility of strongly-types interfaces), we suggest "virtualizing" business objects and manipulating them in a query-like style via unified generic CRUD (Create, Retrieve, Update, Delete) interface. The following subsection describe the suggestions in detail.

3.1 Virtualizing Business Objects

Business objects are the corner stone of BOQL. An important difference to existing object-centered APIs is that in BOQL business objects are virtualized or not materialized. Because BOQL is targeted at main-memory-based ERP systems, there is no need for materializing the objects: all required data always resides in main memory and thus can be directly accessed. Therefore, instead of pulling the data from the database, caching it on the application server and materializing the objects from the cache, the objects can be reconstructed on the fly by resolving high-level data model into physical locations (see Figure 1). This means that whenever a given attribute is required the system calculates its location in the system's address space and performs required manipulation via the storage engine.

The schema resolution is done by business object engine (BO engine) using the metadata (system catalog), which describes the structure of business objects, e.g. their attributes and associations. In its simplest implementation BO engine maps the attributes and associations of business objects to the primitives of the underlaying storage engine (record collections and referential constraint). This, however, is not the best design, as it creates dependency of the BO engine on the storage engine. A much better design is keeping the two engines separated by a storage-independent abstraction layer. This permits the business object engine to work with multiple storage engines using the same API. We called this layer object view infrastructure (OVI).

Figure 2 illustrates the relationships among memory-resident database, storage engine, OVI, BO engine and applications. An application issues a BOQL query that is dispatched to the query engine. The engine parses the query and creates an execution plan: a sequence of CRUD calls on involved business objects. The plan is than executed and the result is than sent back to the application.

Business object virtualization essentially offer an alternative to object-relational mapping (ORM) as a mechanism of assembling business objects from raw data. In comparison to ORM virtual objects impose less overhead. This advantage is achieved by in-place data manipulation. ORM, in contrast, reconstructs business objects only partially and from replicated data. This results in many network round trips between the database and application server. Moreover, ORM complicates the internals of the ERP system, because of the need of keeping database and application server cache in-sync. Virtualized business object, on the other hand, avoid this overhead altogether, by providing access directly to the original data.

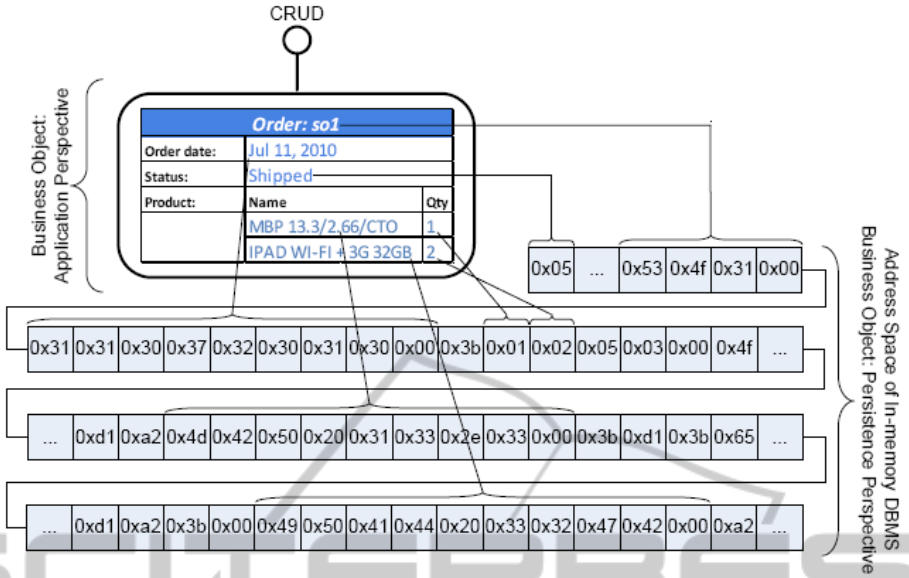

Figure 1: An instance of SalesOrder business object is constructed by ”grouping” memory-resident data and attaching the CRUD interface.

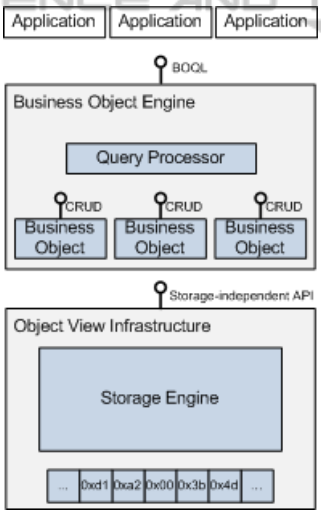

Figure 2: System Architecture.

3.2 Querying Business Objects

In order to support the query-like style of manipulation (and thus overcome the second major deficiency of object-centric APIs - inflexibility) we suggest to unify the interfaces of business objects and design a query engine that would convert declarative queries into a sequence of calls to the limited and well defined number of operations (very much like an SQL engine decomposes queries into relational algebra operations).

The diversity of business objects makes it impossible to inherit them all from a single data type with semantically rich interface. Therefore, we suggest inheriting all business object data types from a class with generic (semantic-free) interface, that features the collections of attributes and associations and CRUD-operations - *Create, Retrieve, Update, Delete*. Such a uniform representation allows introducing a query language for business objects very much like SQL for relations with a difference that a query is translated into a sequence of CRUD-operations, rather than relational algebra operations. Hence, query processing is implemented as follows:

1. **Write Query.** A programmer composes a BOQL query describing what to retrieve from or change in the system and sends the query to the BOQL query processor.
2. **Build Query Tree.** The processor parses the query to detect involved business objects and operations on them and builds a query tree - an internal representation of the query.
3. **Emit Execution Script.** The query processor traverses the query tree and emits an execution script - a sequence of CRUD-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. **Bind Parameters.** Having constructed the call sequence, the processor binds corresponding query tokens to the input parameters of CRUD-operations, e.g. the value of a filter in *where*

clause.

5. **Execute.** The execution script gets executed by business object engine and the result is sent back to the client.

In its essence BOQL performs on-the-fly orchestration of calls to objects' CRUD-operations based on user-defined queries. This orchestration is made feasible by a uniform representation of business objects (in terms of the structure and behavior). In comparison to other object- and service-centric APIs BOQL has an advantage of supporting access and manipulation at any granularity level. At the same time, BOQL does not circumvent business rules (because CRUD-operations control data manipulation and enforce all required constraints), which favors BOQL over the SQL-based APIs. To the query processor the data is seen through the interfaces of business objects. The direct access to the data is prohibited to enforce data integrity and internal business logic implemented by business objects. To access the business data a BOQL query must be issued.

Because the query processor is a single point of access into the system, it is a potential bottle-neck. In order to make the query processor scalable, it has to support multi-threading and allow simultaneous and independent processing of individual queries (inter-query parallelism). Therefore, when the processor receives a query, it acquires a thread from the system's thread pool. The worker thread processes the query as outlined above. In order to further increase the scalability of BOQL, intra-query parallelism can be supported. This can be achieved by parallelizing the execution of CRUD-calls. To figure out what calls to execute at the same time, the query processor analyzes the query tree and issues every call, which does not rely on yet not retrieved data, in a separate thread. Both intra- and inter-query parallelism will not produce any positive effect, unless there is enough hardware capacity available (most importantly the number of CPUs/cores).

3.3 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.

3.4 Integrating BOQL into a Programming Language

In order to improve the productivity of ERP application developers, the effort of issuing BOQL queries from business logic code must be minimized. Therefore, we suggest integrating BOQL with the language used for programming business logic. The integration must be at three levels: syntactic, type system and execution.

Syntactic integration means that the grammar of the chosen programming language must be extended with BOQL grammar. This enables mixing BOQL queries with control statements written in the chosen language. Type system integration means extending the language's type system with business object data types. This makes business objects first-class citizens of the programming language. Integration at execution level (or runtime integration) means that business logic code and data manipulation statements execute

on the same runtime/infrastructure. Strictly speaking, syntactic and type system integration automatically imply runtime integration.

The simplest way of integrating BOQL into a programming language is to cross-compile BOQL queries into this given language. This immediately implies that business object data types and object view infrastructure must be implemented in the very same language (or in compatible ones).

3.5 Prototype System

The current subsection demonstrates a possible implementation of the concepts presented in the paper. In particular, a small ERP system featuring six business objects[1] has been developed. The ERP exposes BOQL as a single data manipulation API. The prototype uses Berkeley DB, configured as an in-memory data store, as its storage engine. The system assumes row-oriented data organization. Because Berkeley DB is a key-value store, it does not natively support row layout. Therefore, the prototype takes the burden of reconstructing rows out of key-value pairs. This functionality is provided by object view infrastructure. As for BOQL query execution, two models are supported: interpretation and cross-compilation. All components of the system are developed in C++. Obviously, queries that are cross-compiled are cross-compiled into C++. The prototype system also integrates BOQL into a programming language (C++) as described in subsection 3.4. For convenience, the programs mixing BOQL and C++ statements are called to be written in Business C++ (BC++). Hence, BC++ is a language integrating C++ control statements (e.g. function calls, assignments, arithmetic expressions, loops, branching, etc.) and BOQL data manipulation statements by providing a single type system and runtime environment for both types of statements. BC++ programs are translated into C++, compiled with GNU g++, linked against the business object engine and later are executed inside its address space (the closest possible to data) and thus become its integral part. The rest of the subsection gives more details on the prototype.

Using Flex (lexer generator) and Bison (parser generator) tools a parser for Business C++ has been generated. Its main purpose is to recognize and process BOQL queries embedded in C++ programs. Therefore, the parser ignores any constructs other than BOQL queries and emits them unchanged (as they are) directly to the output file[2]. For BOQL constructs the parser builds an abstract syntax tree (we call it query tree). If the query stands alone (is not part of any program), the generated tree is handed over to query interpreter. The interpreter traverses the tree and issues CRUD-calls on involved business objects. If the query is embedded into a C++ program, it must be cross-compiled into C++ and merged with the program. Therefore, for an embedded query the query tree is handed over to the C++ code generator (cross-compiler). The later generates a C++ program performing CRUD-calls on business objects. Then, the query code is merged with the rest of the program and resulting code is compiled with a native C++ compiler (GNU g++). The developed parser is capable of processing BOQL data manipulation (DML) statements and data definition (DDL) statements. That is, the system allows not only querying business objects, but also defining them. There is, however, one limitation. DDL statements can only be interpreted, therefore, they cannot be embedded into a C++ program.

The following steps describe the process of working with the system. For demonstration purposes we have selected a dunning use-case.

Define Business Object Data Model. Before working with the system its data model must be defined. This can be done with *Create Type* statement. The following code defines sales order business object data type. The rest five types are defined in the same way.

```
Create Type SalesOrder {
attributes:
  string id,
  DateTime date
associations:
  Customer Customer,
  OrderLineItem Items
};
```

Each *Create Type* statement leads to creation of two C++ classes: business object data type and object view class. The first class implements data schema, whereas the second one maps the schema to the primitives of Berkeley DB. The following code demonstrates classes generated for SalesOrder type[3].

```
class SalesOrderBo : public BusinessObject {
public:
  SalesOrderBo(BoView *owner, char *id);
  string id();
  DateTime date();
  CustomerBo *Customer();
  vector<OrderLineItemBo*> &Items();
};
...
```

[1] customer invoice, sales order, order line item, customer, product and address

[2] The reason is simple: C++ constructs must be compiled with native C++ compiler. So parsing them at this stage is redundant.

[3] for the sake of brevity non-conceptual details were omitted

```
string
SalesOrderBo::id() {
  string res;
  char *rawVal = boView->getAttributeRaw(0, getKey());
  res.assign(rawVal);
  return res;
}
...

class SalesOrderView : public BoView {
public:
  SalesOrderView(DbEnv *env);
  ~SalesOrderView();
  void associate(DbEnv *env);
  //CRUD
  ...
  vector<SalesOrderBo*> &Retrieve();
  ...
private:
  Db *OrderLineItemInd;
};
...
vector<SalesOrderBo*> &SalesOrderView::Retrieve() {

  vector<SalesOrderBo*> *res = new vector<SalesOrderBo *>();
  vector<char*> id = GetIds();
  SalesOrderBo *bo;
  for (int i = 0; i < id.size(); i++) {
    bo = new SalesOrderBo(this, id[i]);
    res->push_back(bo);
  }
  return *res;
}
//Retrieve IDs from Berkeley DB
vector<char*> &BoView::GetIds() {

  vector<char*> *res=new vector<char*>();
  Dbc *cur; Dbt key_, value_;
  collection->cursor(NULL, &cur, 0);
  ...
  int ret;
  while((ret = cur->get(&key_, &value_, DB_NEXT))==0) {
    res->push_back((char*)key_.get_data());
    ...
  }
  ...
  return *res;
}
```

Mount Business Objects Data Types. Once the classes have been generated they are compiled and dynamically loaded into the address space of the system. Then, the classes are registered in the system's catalog, which makes the data types available to the query processor. Mounting data types is done automatically by the system.

Inserting Data. Having defined business objects, they must be populated with data. This step is omitted, as we generated test data.

Write Business C++ Programs. Once the data model and actual data are ready a programmer may start developing their Business C++ code.

```
void RunDunning(Currency email,
                Currency sms, Currency call) {

  DateTime curDate = System.CurrentDate;
  //retrieve overdue invoices
  CustomerInvoice[] inv = SELECT OBJECTS
    FROM CustomerInvoice as CI
    WHERE CI~Status != Status.Paid AND CI~dueDate < curDate;
  //send reminders
  for(int i=0; i<inv.size; i++) {
    if (inv[i]~Amount <= email)
      EmailAlert(inv[i]);
    else if (inv[i]~Amount <= sms)
      SmsAlert(inv[i]);
    else if (inv[i]~Amount <=call)
      CallAlert(inv[i]);
  }
}
```

Execute. An application developer may now call the program by issuing the *Execute* instruction with corresponding input parameters.

Hence, Business C++ offers two advantages over the SQL stored procedures:

- BC++ integrates business object hierarchy of a given ERP system into C++, which allows mixing control flow statements with data manipulation statement;
- BC++ compiles (translates) into a C++ program (the language of storage engine).

4 FUTURE WORK

The presented techniques will result in a higher workload for the DBMS. Therefore, the scalability of a database system got high attention in the current research. A number of approaches on database scalability based on horizontal partitioning and data distribution within a cluster of nodes have been studied (Boral et al., 1990; DeWitt et al., 1990; Teeuw and Blanken, 1993). In order to benefit from the computing resources available in a database cluster, conventional query processing approaches must be revised. In particular, operators within a query execution plan have to be assigned to processing nodes according to the nodes' data. Those distributed operators can require additional communication based on the required synchronization and collection of their results (Mehta and DeWitt, 1997).

Because distributed query processing is a challenging research topic in itself, contributing to it is

out of the scope of the current work. The main contribution to this research area will be devising formal models for optimizing database partitioning strategies, that is, deciding at what parts a database must be split.

5 CONCLUSIONS

The feasibility of in-memory DBMSs and their tremendous performance advantage over the conventional DBMSs in ERP systems market sector provoked us to rethink the distribution of workload in an ERP system. Orders of magnitude performance advantage of in-database operations suggests shifting more business logic operations inside the database. This requires the database to provide a semantically richer data model and API than currently available. In this paper we presented an API, called business object query language, that we think better suits the needs of ERP application developers than any of the currently employed APIs.

The main idea of BOQL is to provide a system that groups together different pieces of memory-resident data and casts the resulting constructs to specific business object data types. That is, BOQL is essentially a system that resolves high-level abstractions (business objects) into low-level storage primitives that are manipulated via the storage engine's API. We also demonstrated how BOQL can be integrated into an object-oriented programming language by means of cross-compiling BOQL queries into the target language.

REFERENCES

Bernstein, P. A., Pal, S., and Shutt, D. (1999). Context-based prefetch for implementing objects on relations. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 327–338.

Boral, H., Alexander, W., Clay, L., Copeland, G., Danforth, S., Franklin, M., Hart, B., Smith, M., and Valduriez, P. (1990). Prototyping bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):24.

Borovskiy, V., Enderlein, S., and Zeier, A. (2009). Generic web services - extensible functionality with stable interface. In *IEEE International Conference on Web Services*.

DeWitt, D., Ghandeharizadeh, S., Schneider, D., Bricker, A., Hsiao, H., and Rasmussen, R. (1990). The gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, pages 44–62.

Grund, M., Krueger, J., and Zeier, A. (2008). Declarative web service entities with virtual endpoints. In *Proceedings of the IEEE International Conference on Services Computing*.

Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E. P. C., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., and Abadi, D. J. (2008). H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499.

Legler, T., Lehner, W., and Ross, A. (2006). Data mining with the sap netweaver bi accelerator. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 1059–1068. VLDB Endowment.

Manegold, S., Boncz, P. A., and Kersten, M. L. (2000). Optimizing database architecture for the new bottleneck: memory access. *The VLDB Journal*, 9:231–246.

Mehta, M. and DeWitt, D. (1997). Data placement in shared-nothing parallel database systems. *The VLDB Journal*, 6(1):53–72.

Resende, L. (2007). Handling heterogeneous data sources in a soa environment with service data objects. In *ACM SIGMOD international conference on Management of data*.

Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N., and Helland, P. (2007). The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 1150–1160. VLDB Endowment.

Teeuw, W. and Blanken, H. (1993). Control versus data flow in parallel database machines. *IEEE Transactions on Parallel and Distributed Systems*, 4(11).