An SQL interface for querying a program's objects

Marios Fragkoulis[a,*], Diomidis Spinellis[a], Panos Louridas[a]

[a]*Department of Management Science and Technology, Athens University of Economics and Business, Patision 76,* GR-*104 34 Athens, Greece*

Abstract

Query facilities typically ship as part of database management systems or, sometimes, bundled with programming languages. Issuing interactive ad-hoc queries on program data is tough. For object-oriented applications, database management systems impose an expensive model transformation; general purpose programming languages lack an interpreter and/or an expressive query language for manipulating such queries.

This work presents a method and an implementation for mapping an arbitrary application's object-oriented model into a queryable relational one. The Pico COllections Query Library (PiCO QL) uses a domain specific language to define a relational representation of object-oriented data structures and a parser-generator to implement an SQL interface. It then carries out queries written in SQL against C++ program objects. PiCO QL queries are issued interactively and are type safe. The paper demonstrates the library's usefulness on three large C++ projects. PiCO QL enhances query expressiveness and boosts productivity compared to querying C++ objects via traditional C++ programming constructs.

Keywords: SQL, query, main-memory, object, C++

*Department of Management Science and Technology, Athens University of Economics and Business, Patision 76, GR-10434, Athens, Greece tel: +30 2108203370

Email addresses: mfg@aueb.gr (Marios Fragkoulis), dds@aueb.gr (Diomidis Spinellis), louridas@aueb.gr (Panos Louridas)

[0]Abbreviations:
PiCO QL: Pico COllections Query Library
HLQL: High Level Query Languages

1. Introduction

The interaction between applications and database management systems (DBMS) has been studied intensively (Reese, 2000; Sanders, 1998). DBMSs complement applications with an API and a standard query language to manage data. They offer an important set of properties, namely atomicity, consistency, isolation and durability (known as ACID) and expressive, powerful views on database data. Outsourcing an application's data management to a DBMS is the typical design option, but one size does not fit all (Stonebraker and Cetintemel, 2005).

There are many applications that do not require a fully-fledged database management system. Such applications either do all their processing completely online, or store their data using bespoke file formats. Program data may be stored in data structures provided by the programming language, such as the C++ STL and *Java.util* containers. Containers are an efficient vehicle for storing complex objects and running algorithms like set operations, but they do not offer an easy way to perform ad-hoc queries on the program objects stored in them. Importing data into a DBMS just to improve querying introduces a superfluous dependency and overhead. PiCO QL solves this problem by providing an SQL interface for querying program objects.

The problem addressed in this paper is different to interfacing programs with DBMSs. In PiCO QL the program memory space acts as the database, and the aim is to provide database views of the program data. By contrast, when using a DBMS, programs have their data stored in a database and aim to bring database views into the program context.

The contribution of this work is:

- a brief classification of query languages and interfaces, a synopsis of tools for querying program data through an external interface and a synopsis of tools for querying (un)structured file data through an external interface,
- a method for mapping the object-oriented model into a queryable relational one,
- a demonstration of the method's validity through the implementation and evaluation of PiCO QL, a library that supports interactive SQL queries against C++ program objects.

2. Related work

This paper describes an implementation of an external SQL interface to query program objects. To do that, we combine object-relational mapping and object query evaluation techniques. Our work has limited affinity with query languages and techniques for querying object-oriented data (see section 2.1). It relates to approaches that provide ad-hoc queries against data in program memory space (see section 2.2), and to query languages that offer ad-hoc queries to file-based (un)structured data (section 2.3).

2.1. Query languages and techniques for querying OO data

In this section we examine popular query languages and interfaces for querying OO data, and present PiCO QL's position in this context (section 2.1.1). We also examine techniques for query evaluation that OO query languages adopt (section 2.1.2), and object-relational mapping techniques that enable interaction between an object-oriented application and a relational database management system (section 2.1.3).

2.1.1. Query languages and interfaces for querying OO data: a classification

To compare this work with query languages and interfaces for querying OO data it is useful to partition the latter in four classes using a 2×2 matrix, with columns differentiating data location (external database or programming language data structures) and rows differentiating the query interface (external or part of the programming language). Figure 1 shows this classification.

The first class includes query languages that function within the program scope and interface with data that reside in a database. Examples include JPQL (Keith and Schincariol, 2009), ScalaQL (Spiewak and Zhao, 2010) and LINQ to SQL (Meijer et al., 2006). As already mentioned in section 1, PiCO QL deals with a different problem compared to query languages of this class, namely interfacing through a relational interface to data inside the program.

In the second class we classify DBMS interfaces that offer interactive querying of database data. All modern database management systems offer such an interface.

A third class groups query languages that function within the program scope and support queries against data collections residing in the program's memory space. Examples include LINQ to objects (Meijer et al., 2006), JQL (Willis et al., 2006), RelC (Hawkins et al., 2011) and others (Schwartz

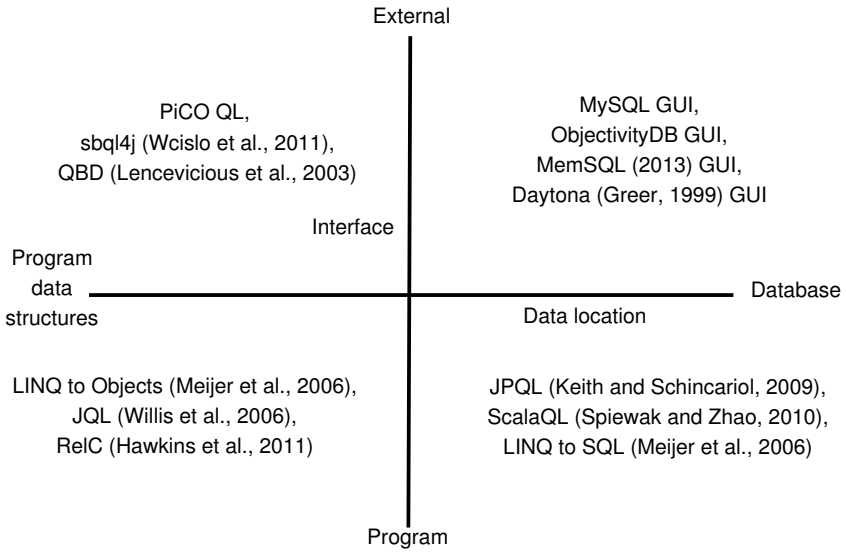


Figure 1: Object-oriented query language classes

et al., 1986; Adamus et al., 2008). Query languages in this class query elements residing in the program memory space, like PiCO QL, but have a different focus and aim. They perform efficiently specific programming tasks and extend the programming language focusing on data retrieval through queries fixed at compile time. By contrast, PiCO QL provides a data analysis tool that focuses on runtime query flexibility and expressiveness. Some of the referenced query languages in this class do attempt to offer an interactive query interface (see section 2.2.2), bringing them closer to our work.

PiCO QL belongs in a fourth class, which lies opposite to query languages interfacing programs with database systems. It is representative of a different class of query languages that will be described in section 2.2.

2.1.2. Object-oriented query evaluation techniques

Queries in object oriented query languages typically leverage path expressions (Frohn et al., 1994) in the execution mechanism. Path expressions are a core aspect of most object-oriented query languages. These support a declarative dialect for issuing nested object-oriented queries. Queries can reference literals, objects, and collections nested in arbitrary depth.

2.1.3. Object-relational mapping techniques

Fundamental issues in object-relational mapping concern the mapping of programming language classes, associations (*has-a, many-to-many*), inheritance (*is-a*) and polymorphism to relational constructs.

Under most approaches classes become relational tables and non-scalar values within classes, i.e. associations, give their place to relationship instances, i.e. primary key-foreign key chains between relational tables. *Many-to-many* associations require an intermediate table for tracking the association instances between the two tables.

Three approaches have been proposed to implement a relational mapping of inheritance and polymorphism.

1. Include the whole class hierarchy in a single table, thus each column maps to each attribute of the class hierarchy and an extra column identifies the object type.
2. Each table maps to a class, abstract or concrete, so that the full information on an object is obtained by joins on the tables up the hierarchy.
3. Map each concrete class to a table containing all the attributes (both own and inherited).

Object-relational mapping techniques are implemented by object-relational mapping frameworks. Two popular ones are Hibernate for Java (Bauer and King, 2006) and Microsoft ADO .NET Entity Framework for .NET (Melnik et al., 2007).

2.2. Ad-hoc queries to program data

Ad-hoc queries to program data is an area containing diverse lines of research, including PiCO QL. It includes main memory D(B)MS that offer a query interface to database data (section 2.2.1), query languages that leverage programming through querying and attempt to support ad-hoc queries (section 2.2.2), and domain specific ad-hoc queries to program data (section 2.2.3).

2.2.1. Ad-hoc queries to main memory D(B)MS

Main memory query processing employs pointers for cross-referencing of data structures, including object associations and foreign keys to relational table tuples (Lehman and Carey, 1986). In these cases pointers drive the use of precomputed joins where, for example, a foreign key is substituted with the tuple(s) it points to.

Main memory OO DBM systems like MemSQL (2013), support SQL queries through an external query interface. MemSQL uses lock-free data structures in memory, translates SQL queries to C++ code for efficiency, swaps data to disk after a transaction succeeds, and behaves exactly like MySQL, with which it is wire-compatible.

AT&T's Daytona (Greer, 1999) data management system with its high level fourth generation query language Cymbal also fits in this category. Cymbal is a powerful multi-paradigm language, which includes ANSI 89 SQL as a subset. Daytona is based on a code-generation architecture, like PiCO QL. Ad-hoc queries in Cymbal translate into C programs complete with a makefile, compile, and execute against data stored in standard UNIX filesystems. Cymbal provides containers optimized for in-memory computation, which can also support applications required to operate in main memory at all times.

2.2.2. Ad-hoc queries to object collections

Some object-oriented query languages for in-memory object collections attempt to offer ad-hoc queries. JQL (Willis et al., 2006) and OQL/C++ (Cattell and Barry, 2000) provide limited support for interactive queries; *sbql4j* (Wcislo et al., 2011) promises support for interactive queries in the future.

JQL supports dynamic queries against Java collections. The query evaluator can accept an abstract syntax tree at run time and execute the query with the cost of runtime type checking. Although flexible queries can be constructed from user input, there is always the cost of programming the conversion of an untyped query expression to a typed one.

OQL/C++ supports queries against C++ data structures. Since queries take the form of untyped strings, they can be input from an external interface. Query input parameter types, however, are checked at runtime and type violations trigger an error exception. Each query's result is specified as a parameter to the function that executes it. An error exception is generated if the actual result type differs from the specified one.

sbql4j consists of a prototype implementation of SBQL (Adamus et al., 2008) for Java. SBQL, the Stack Based Query Language, is the query language for a Stack Based Architecture (SBA). SBA conceptually and semantically unifies querying and programming by providing orthogonal data/object construction, orthogonal language constructs, type safety, and advanced programming abstractions, like views. Queries in *sbql4j* undergo compile-time analysis. Ad-hoc queries have been proposed as a future extension. These

require that the query language type checking mechanism collaborates with Java's run time type checking mechanisms.

2.2.3. Domain specific ad-hoc queries to program objects

Lencevicius et al. (2003) introduced the query-based approach to debugging (QBD) and a corresponding tool. The debugger demonstrates capabilities for dynamic and on-the-fly queries in Java using load-time code instrumentation. A custom class loader generates and compiles custom debugger code. Queries expose object state and object relationships. The debugger has the ability to gather plain statistical data but does not support sophisticated operations like aggregations, nested queries, and views offered by general purpose query facilities.

PTQL (Goldsmith et al., 2005) is a query language for program traces. It queries the execution trace seeking to match a context-sensitive pattern of events in Java programs. PTQL, an SQL-like language, adopts a relational data model. It performs online query evaluation, but acts upon sequences of events, not object data.

2.3. Ad-hoc queries to file based (un)structured data

In recent years, the exponential growth of (un)structured datasets drove the emergence of the map-reduce programming paradigm (Dean and Ghemawat, 2008). Despite effortless parallelism, programming in a map-reduce fashion using procedural statements hindered its full potential. High level query languages (HLQL) like Pig Latin (Olston et al., 2008), HIVEQL (Thusoo et al., 2010) and JAQL (Beyer et al., 2009) came forward to address this (Stewart et al., 2011). HLQLs like SCOPE (Chaiken et al., 2008), which defines an extensible relational model, and the query language provided by Dremel (Melnik et al., 2010), which operates on a columnar storage representation share the same objectives as map-reduce and complement it.

The above HLQLs are languages for ad-hoc analysis of datasets. Except for JAQL, HLQLs resemble SQL. JAQL is a functional higher-order query language; it offers support for filter, transformation, aggregation, sort, group by, and join operations. SCOPE is a scripting language whose syntax strongly resembles SQL, and, in addition, it models data as sets of rows with strictly typed columns. It is highly extensible with user-defined operators and functions. SCOPE interacts with the Cosmos (Chaiken et al., 2008) execution environment, which is more flexible than map-reduce in that it can handle any task expressible as an acyclic data flow graph. Dremel provides an

SQL-like language that executes ad-hoc queries on a columnar representation of nested data without using a map-reduce layer. Its architecture, a serving tree, is widely used in distributed search engines. Pig Latin is a data flow language that blends SQL with procedural programming. In Pig Latin, queries consist of a sequence of steps with each step declaring a query operation at a high level (SELECT etc.), similarly to an SQL clause. HIVEQL is a declarative query language; it implements a subset of SQL and, mostly, adopts its syntax. It supports nested queries in the FROM clause, aggregations, joins, group-by clauses, views, and functions on data types. According to Stewart et al. (2011), JAQL is Turing-complete while HIVEQL and Pig Latin are computationally equivalent to SQL. The latter become Turing-complete with the introduction of user-defined functions.

Querying datasets stored in files is not a new idea. Data management interfaces like ODBC (Sanders, 1998) have supported SQL queries over CSV files for years.

3. A relational interface for OO data

Our method for exposing the object-oriented data model through a relational interface addresses two challenges: first how to provide a relational representation of object-oriented data structures; secondly how to map SQL queries to main-memory object-oriented data structures through their relational representation. The key points of the design that address these challenges include (a) rules for creating a relational representation out of object-oriented data structures, (b) a domain specific language (DSL) for specifying relational representations and access information of object-oriented data structures, (c) an evaluation method for relational queries in this object-oriented environment, and (d) the formal description of a new operator introduced to achieve the mapping of object associations to a relational representation. We conclude the section with a synopsis of the steps required to embed the relational interface into an application.

3.1. Relational representation of object-oriented data structures

The problem of transforming object-oriented data to their relational counterpart, and vice versa, has been studied thoroughly in the literature (Bauer and King, 2006; Melnik et al., 2007). We address a different but related, problem: how to represent object-oriented data structures in relational terms. Providing a relational interface to object oriented data without storing them

in a relational database management system is not straightforward; the issue at hand is not the transformation of data from object structures into relational structures, but the representation of data in different models. In other words, we do not transform the object data; instead we want to provide a relational view on top of it. Issues that emerge in bidirectional transformations (Czarnecki et al., 2009) between data models, such as the O-R cross-metamodel data mapping, are not examined.

In this work the underlying data model is solely object-oriented. This leverages object-oriented features like inheritance and subtype polymorphism in queries. To do that, however, we must solve the representation mismatch between relations and objects. Relations consist of a set of columns that host scalar values, while objects form graphs of an arbitrary structure.

PiCO QL provides a relational representation of object-oriented data structures in the form of virtual tables (The SQLite team, 2013) supported by SQLite (Owens, 2006), an embeddable database engine. This includes *has-a* associations between objects (section 3.1.1), *many-to-many* associations between objects (section 3.1.2) and *is-a* associations and subtype polymorphism (section 3.1.3).

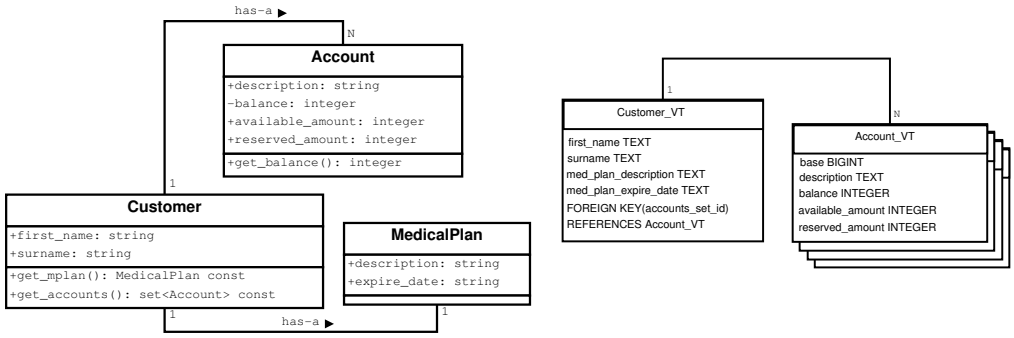
3.1.1. Mapping has-a associations

Has-a associations are of two types: *has-one* and *has-many*. Here we describe how these are represented. Let the *containing* object be an object with some contents and the *contained* be those contents. *Has-one* associations include built-in primitive values, references to primitives, objects and references to objects. These are represented as columns in a virtual table that stands for the containing object. *Has-many* associations include collections of contained objects and references to collections of contained objects. These are represented by an associated table that stands for the collection of contained objects. Although the associated table's schema is static, *the contents of the associated table are specific to the containing object instance*: each instance of the containing instance has distinct contents.

Figure 2(a) shows the class diagram of a trivial object-oriented data model. Figure 2(b) shows the respective virtual table schema. On the schema, each record in the customer table (Customer_VT) represents a customer. A customer's associated medical plan has been included in the customer's representation: each attribute of the MedicalPlan class occupies a column in Customer_VT. In the same table, foreign key column *accounts_set_id* identifies the set of accounts that a customer owns. A customer's account

information may be retrieved by specifying in a query a join with the account table (Account_VT). By specifying a join with the account table, an instantiation happens. The instantiation of the account table is customer specific; it contains the accounts of the current customer only. For another customer another instantiation would be created. Thus, multiple instances of *Account_VT* implicitly exist in the background as Figure 2(b) shows.

Figure 2: One-to-many association



(a) Class diagram (b) Virtual table schema

In PiCO QL we provide each virtual table representing a nested data structure with a column named *base*, which takes part in *has-a* associations. The one side of the association is rendered by a foreign key column, which identifies the contents of an associated table as shown in the previous example, and the other side is rendered by the associated table's *base* column, which fulfills an appropriate instantiation. The *base* column is instrumental both for mapping associations to a relational representation and for mapping the relational query evaluation to the underlying object-oriented environment (Section 3.3).

3.1.2. Mapping many-to-many associations

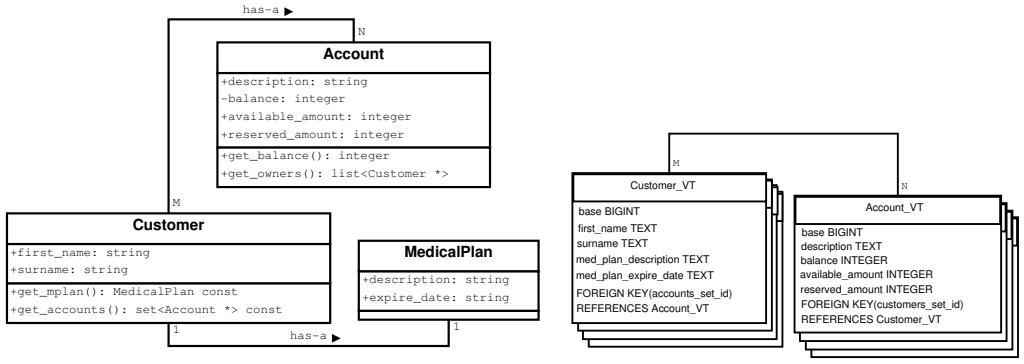
Many-to-many associations require no special treatment; they are treated similarly to *has-many* associations. Continuing with the bank application example, suppose that, in addition to the above specification, an account can have many owners (Figure 3). The relationship can be described as a *has-many* association from both sides, that is, account to bank customer and vice versa. The effect in the virtual table schema is multiple instantiations for

Customer_VT as well, since it is now possible to identify the bank customers that own a specific account.

In the relational model, a *many-to-many* relationship requires an intermediate table for the mapping.

In PiCO QL virtual tables provide a relational representation to an application's data structures, but are only views on the data. For each instance of a Customer (say John_Doe) a distinct Account_VT (say John_Doe_Account_VT) virtual table is instantiated. Similarly, for each instance of an Account (say Saving Account) a distinct Customer_VT (say Saving_Account_Customer_VT) is instantiated. Section 4.2 discusses the amount of effort and computational cost in creating virtual table instantiations.

Figure 3: Many-to-many association



(a) Class diagram (b) Virtual table schema

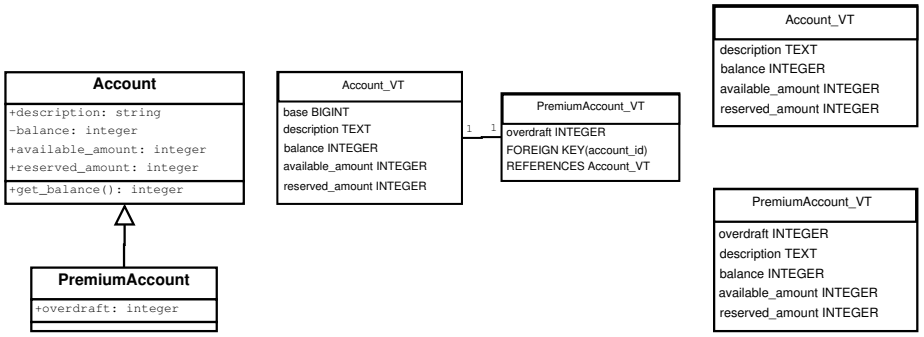
3.1.3. Mapping is-a associations

The support of *is-a* associations in the object-oriented paradigm provides powerful features, namely inheritance and subtype polymorphism. PiCO QL offers two ways to incorporate OO inheritance and subtype polymorphism addressable at a relational representation of data structures; see Figure 4(a) for an example inheritance hierarchy. These ways correspond to ones supported by RDBMSs.

First, it is possible to represent each class in the inheritance hierarchy as a separate virtual table and use a relationship to link them – see Figure 4(b), following the *table to class* mapping approach. Second, it is possible to include inherited members as columns in each of the subclasses represented as

virtual tables – see Figure 4(c), following the *table to concrete class* mapping approach.

Figure 4: Inheritance and subtype polymorphism support

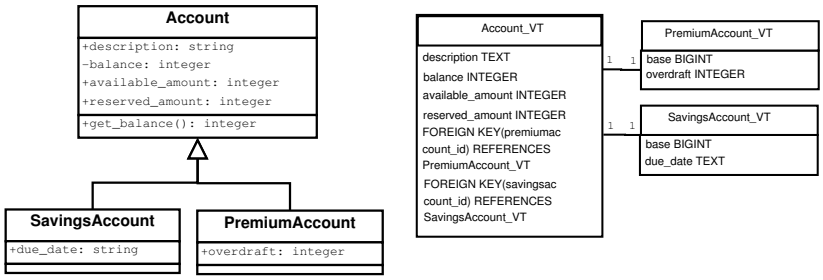


(a) Class diagram (b) Table to class mapping (c) Table to concrete class mapping

For polymorphic containers care must be taken in representing their contents that involve subtypes of the container element type. Suppose we represented a polymorphic container of accounts, that is each element could be a reference to a savings account or premium account, as in Figure 5(a). Virtual table Account_VT — see Figure 5(b), which represents the container of accounts, includes columns that map to members of Account type. This way, basic account information can be retrieved directly from Account_VT. Virtual table PremiumAccount_VT includes columns that map to members of type PremiumAccount. Similarly virtual table SavingsAccount_VT includes columns that map to members of type SavingsAccount. A relationship instance links the virtual table representing the base class with the virtual table representing a derived class. Consequently premium, savings account information can be retrieved from the virtual tables representing the derived classes through the relationship instances.

During a query (Listing 8), columns that map to derived type members may be accessed by issuing joins to link the relational representation of the base class to the relational representations of the derived classes. Joins between SQLite virtual tables take the form of left outer joins; hence join operations trigger checks in the background to match a container element's derived type against the type represented by a derived class' relational representation. In the case, say, of an account container element which is a reference to a PremiumAccount object instance, the type check as a result

Figure 5: Full support of polymorphic containers



(a) Class diagram (b) Virtual table schema

of joining with PremiumAccount_VT will succeed, since PremiumAccount_VT represents type PremiumAccount, while the type check resulting from joining with SavingsAccount_VT will fail and the query will terminate with a misuse error message. Type checks ensure type consistency for each container element. In the result set, the columns of the relational representations of other derived classes than the one the element belongs are populated with null values.

3.2. A Domain Specific Language for defining relational representations of object-oriented data structures

PiCO QL provides a DSL for describing the mapping of the OO data into a relational model. The mapping is performed in two steps: a) *struct view* definitions describe a virtual table's columns and b) *virtual table* definitions link a *struct view* definition to a program's data structure type. Together they compose a relational representation definition. The DSL's syntax is formally described in Listing 1 using Backus-Naur Form (BNF) notation.

Listing 1: DSL syntax in BNF notation

```
; Virtual table definition
<virtual_table_def>      ::=   'CREATE VIRTUAL TABLE' <virtual table name>
                               'USING STRUCT VIEW' <struct view name>
                               ['WITH REGISTERED C NAME' <base variable>]
                               'WITH REGISTERED C TYPE' <struct_type>
                               ['USING LOOP' <loop_variant>] '$'

<struct_type>            ::=   <container> | <object> | <struct> |
                               <primitive_data_type> ['*']
```

<container> ::= (<container_class> '<' <struct_type>
; if associative container
[',' <struct_type>] '>' ['*']) | <C_container>

<container_class> ::= <stl_class> | <other_class>

<stl_class> ::= 'list' | 'vector' | 'deque' | 'set' | 'multiset' | 'map' | 'multimap'

<other_class> ::= <SGI forward container concept compatible>

<C_container> ::= <d_type> | <struct> ':' <d_type>
; For C linked lists and C arrays respectively. The loop variants for these
; containers can be customized by the USING LOOP directive
; which may include macros. The latter can be defined at the top of the DSL
; description.

<d_type> ::= <struct> | <primitive_data_type> ['*']

<struct> ::= ['struct'] <struct name> ['*']

<primitive_data_type> ::= 'int' | 'string' | 'double' | 'char' | 'float' | 'real' | 'bool' | 'bigint'

<object> ::= <class name> ['*']

<loop_variant> ::= <user defined loop variant>

; Struct view definition
<struct_view_def> ::= 'CREATE STRUCT VIEW' <struct view name> '('
<column_def> {',\n' <column_def>} ')$'

<column_def> ::= <primitive_column_def> | <struct_column_def> | <struct_view_inclusion>

<primitive_column_def> ::= <column name> <primitive_data_type> 'FROM'
<access_statement>

<struct_column_def> ::= 'FOREIGN KEY(' <column name> ') FROM'
<access_statement> 'REFERENCES'
<virtual table name> ['POINTER']

<access_statement> ::= <valid C++ expression> | 'self'

```
<struct_view_inclusion>   ::=   'INCLUDES STRUCT VIEW' <struct view name>
                                ['FROM' <access_statement> ['POINTER']]

; Standard relational view definition
<rel_view_def>            ::=   <ANSI 92 SQL standard> '$'
```

3.2.1. Struct view definition

Struct view definitions (Listings 2 – 5) describe the columns of a virtual table. They resemble relational table definitions. Struct view definitions include the struct view's name and its attributes. Each attribute description contains the essential information for defining a virtual table column.

Column definitions are of two types, data column definitions and special column definitions for representing *has-a*, *many-many* and *is-a* object associations. Data column definitions include the column's name, data type, and access path, that is, a C++ expression that retrieves the column value from the object. Special column definitions include the column's name, access path and associated virtual table. Two kinds of special column definitions are supported, *foreign key* definitions (Listings 2, 3) and *struct view inclusion* definitions (Listings 4, 5).

Listing 2: Struct view definition - *has-a* relationship (Figure 3)

```
CREATE STRUCT VIEW Account_SV (
  description TEXT FROM description,
  balance INTEGER FROM get_balance(),
  available_amount INTEGER FROM available_amount,
  reserved_amount INTEGER FROM reserved_amount,
  FOREIGN KEY(customers_set_id) FROM get_owners()
                                REFERENCES Customer_VT)
```

The foreign key column definition (Listings 2, 3) supports relationships between virtual tables that represent a *has-a* (Listing 2) or an *is-a* (Listing 3) association between the underlying OO data structures. A foreign key specification resembles its relational counterpart except that referential constraints are not checked in this context and no matching column of the referenced table is specified. This is because the foreign key column matches against an auto-generated column of the referenced virtual table, the *base* column. The *base* column does not appear in relational representation definitions because the DSL parser-compiler can understand when the column is required and generates the appropriate code for it.

Listing 3: Struct view definition - *is-a* normalization (Figure 4(b))

```
CREATE STRUCT VIEW PremiumAccount_SV (
  overdraft INTEGER FROM overdraft,
  FOREIGN KEY(account_id) FROM self REFERENCES Account_VT)
```

Listings 3 and 4 illustrate the supported inheritance mapping options in terms of the DSL. Listing 3 shows how to represent each class in the inheritance hierarchy as a separate virtual table (*table per class* mapping) and using a relationship to link them. *self* is a language keyword that denotes an empty access path. It functions as a placeholder and is used to support an inheritance mapping through a relationship instance. The base object's identity is adequate information for the mapping in these cases. On the other hand, Listing 4 shows support for including inherited members as columns in each of the subclasses represented as virtual tables (*table per concrete class* mapping).

Listing 4: Struct view definition - *is-a* inclusion (Figure 4(c))

```
CREATE STRUCT VIEW PremiumAccount_SV (
  overdraft INTEGER FROM overdraft,
  INCLUDES STRUCT VIEW Account_SV) % Struct view inclusion
```

Including relational representations into others is useful for representing not only *is-a* but also *has-a* associations inline (Listing 5). Such is the case with a medical plan included in a customer's relational representation in Figure 3.

Listing 5: Struct view definition - *has-a* inclusion (Figure 3(b))

```
CREATE STRUCT VIEW Customer_SV (
  firstName TEXT FROM first_name,
  surname TEXT FROM surname,
  FOREIGN KEY(accounts_set_id) FROM get_accounts()
                                REFERENCES Account_VT,
  INCLUDES STRUCT VIEW MedicalPlan_SV
                       FROM get_mplan()) % Struct view inclusion
```

3.2.2. Virtual table definition

Virtual table definitions (Listing 6) link an object-oriented data structure to its relational representation. They carry the virtual table's name and information about the data structure it represents. Data structure information includes an identifier (C NAME) and a type (C TYPE); the identifier maps the application code data structure to its virtual table representation; the type

must agree with the data structure's programming language type. A virtual table definition always links to a struct view definition through the USING STRUCT VIEW syntax.

Listing 6: Virtual table definition

```
CREATE VIRTUAL TABLE Account_VT
USING STRUCT VIEW Account_SV
WITH REGISTERED C NAME accounts
WITH REGISTERED C TYPE set<Account *>;
```

3.3. Mapping a relational query evaluation to the underlying object-oriented environment

In PiCO QL, the relational representation of an object-oriented data structure comprises one or more virtual tables. Each virtual table in the representation enables access to some part of OO data structure using path expressions (see Listing 12 for an example of the underlying auto-generated routines).

For example, a container of *PremiumAccount* objects could be represented by rendering the *is-a* association between classes *Account* and *PremiumAccount* via a table per class mapping — recall Figure 4(b); then the design would include two virtual tables, one for each class. The virtual table representing the *Account* type provides access to Account members and the virtual table representing the *PremiumAccount* type provides access to *PremiumAccount* members. Member access is provided by path expressions according to the DSL specification.

Virtual tables may be combined in SQL queries by means of join operations (Listings 7, 8). Object-oriented data structures may span arbitrary levels of nesting. Although the nested data structure may be represented as one or more virtual table(s) in the relational interface, access to it is available through a parent data structure only. The virtual table representing the nested data structure (VT_n) can only be used in SQL queries combined with the virtual table representing a parent data structure (VT_p). For instance, one cannot select a customer's associated medical plan without first selecting the customer. If such a query is input, it terminates producing a misuse error message.

A join is required to allow querying of VT_ns. The join uses the column of the VT_p that refers to the nested structure (similar to a foreign key) and the VT_n's base column, which acts as an internal identifier. When a join operation references the VT_n's base column it instantiates the VT_n by setting

the foreign key column's value to the base column. This drives the new instantiation thereby performing the equivalent of a join operation: *for each value of the join attribute, that is the foreign key column, the operation finds the collection of tuples in each table participating in the join that contain that value*. In our case the join is a precomputed one and, therefore, it has the cost of a pointer traversal. The base column acts as the activation interface of a VT_n, and guarantees type-safety by checking that the VT_n's specification is appropriate for representing the nested data structure.

Providing relational views of object-oriented data using a relational query engine imposes one requirement to SQL queries. VT_ps have to be specified before VT_ns in the FROM clause. This stems from the implementation of SQLite's syntactic join evaluation and does not impose a limitation to query expressiveness.

Listing 7: Join query — querying *is-a* and *has-a* associations

```
SELECT * FROM PremiumAccount_VT
JOIN Account_VT
ON Account_VT.base = PremiumAccount_VT.account_id
JOIN Customer_VT
ON Customer_VT.base = Account_VT.customer_id;
```

Listing 8: Join query — querying polymorphic data structures

```
SELECT * FROM Account_VT
JOIN PremiumAccount_VT
ON PremiumAccount_VT.base=Account_VT.premiumaccount_id
JOIN SavingsAccount_VT
ON SavingsAccount_VT.base=Account_VT.savingsaccount_id;
```

In addition to combining relational representations of associated data structures in an SQL query, joins may also be used to combine relational representations of unassociated data structures; this is implemented through a nested loop join. Say the bank regulator requested account information of specific bank customers by providing their full names. After incorporating the customer list in the system and creating its relational representation (IRSCitizen_VT) we could issue the query shown in Listing 9.

Listing 9: Relational join query

```
SELECT * FROM Customer_VT, IRSCitizen_VT
WHERE Customer_VT.first_name = IRSCitizen_VT.first_name
AND Customer_VT.surname = IRSCitizen_VT.surname;
```

3.4. Relational algebra operators

Although relational algebra concerns sets of items, its operators can be applied to arbitrary collections of items. According to Meijer (2011), LINQ also builds on this approach.

PiCO QL introduces a new operator, β,[1] to instantiate virtual tables that represent object collections accessible through others. Let Ξ be a virtual table representing an object collection (the *containing*) and Λ a virtual table representing an object collection (the *contained*) nested within the first collection. The instantiation of Λ relies on its *base* column, which is matched to a corresponding *foreign key* column of Ξ (fk_{base}). For each row of Ξ (say ξ_*) the operator instantiates a virtual table (Λ_*) representing the contained object collection (presented in Sections 3.1.1, 3.1.2). This operation casts the foreign key column value into the contained object collection's type and the object collection starting at this address is instantiated through the virtual table representing it. A foreign key column represents an object association as explained in Sections 3.1.1 and 3.2.1. A formal description is presented in Listing 10.

Listing 10: Formal description of virtual table instantiations

Let β be an operator casting an untyped memory address into a pointer to a primitive type, an object, or a collection of objects,
Ξ be a virtual table instantiating the containing object collection ,
$\xi_1, \xi_2, ..., \xi_\nu$ be rows of virtual table Ξ, that is $\xi_1, \xi_2, ... \xi_\nu \epsilon \Xi$, and
$\Lambda_1, \Lambda_2, ..., \Lambda_\nu, \Lambda$ be virtual table instantiations of the contained object collection
then

$$\beta(\pi_{fk_{base}}(\xi_1)) \rightarrow \Lambda_1,$$

$$\beta(\pi_{fk_{base}}(\xi_2)) \rightarrow \Lambda_2,$$

$$...$$

$$\beta(\pi_{fk_{base}}(\xi_\nu)) \rightarrow \Lambda_\nu,$$

$$\beta(\pi_{fk_{base}}(\Xi)) \rightarrow \Lambda, \quad \text{where} \quad \Lambda = \Lambda_1 \cup \Lambda_2 \cup ... \cup \Lambda_\nu$$

In addition, PiCO QL supports all relational algebra operators as implemented by SQLite (Owens, 2006), that is, the SELECT part of SQL92 excluding right outer joins and full outer joins. Queries expressed using the latter, however, may be rewritten with supported operators (Owens, 2006). For a right

[1]The initial letter of the Greek word $\beta\acute{\alpha}\sigma\eta$ which means base

outer join, rearranging the order of tables in the join produces a left outer join. A full outer join may be transformed using compound queries.

For the most part, query efficiency mirrors SQLite's query processing algorithms enhanced through the following of pointers in memory.

3.5. Embedding PiCO QL in applications

The process for embedding PiCO QL in an application (Figure 6) includes three steps. First, PiCO QL requires a relational representation of the target data structures and data structure information in order to generate code for querying them. These pieces of information are supplied to PiCO QL through its DSL. Then the data structures have to register with PiCO QL through a specific function call and a single function call is required to start the library (Listing 11). Finally, the application makefile has to change to include PiCO QL compile directives.

Listing 11: PiCO QL directives

```
#include "pico_ql_search.h"
using picoQL;
...
pico_ql_register("accounts", &systemAccounts);
pico_ql_serve(8080);
```

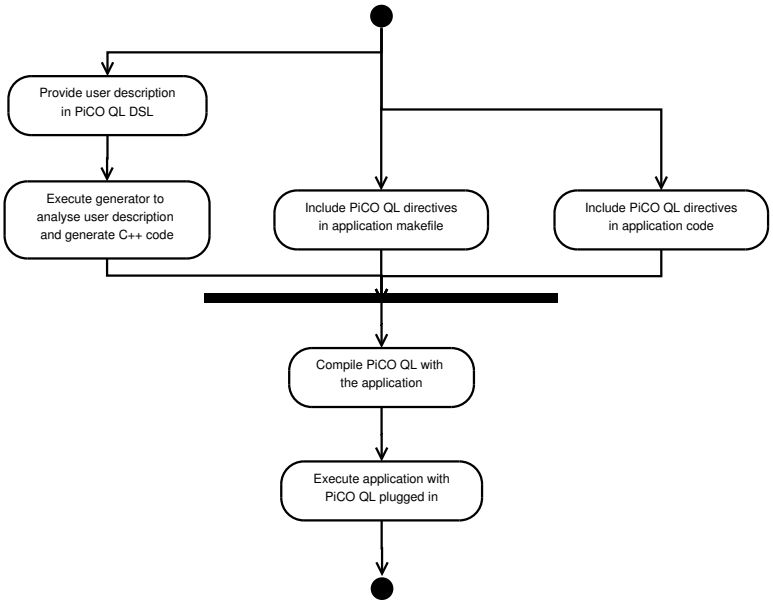
4. PiCO QL implementation specifics

PiCO QL is an SQL query library for C++ applications. The cornerstone of the PiCO QL implementation is a meta-programming technique (section 4.1) devised to (a) create a relational representation of arbitrary C++ program data structures using a relational specification of these data structures, and (b) generate C++ code for querying the data structures through their relational representation. PiCO QL leverages the virtual table module of SQLite to support OO data (section 4.2). The SWILL (Lampoudi and Beazley, 2002) library is used to expose the query service to end users through a web-based interface (section 4.3).

4.1. Generative programming

Our solution to the problem of representing any C++ data structure as a virtual table is based on a user-driven relational specification and a meta-programming technique, generative programming. Specifically, a parser-generator analyzes relational specifications and C++ program data structure

Figure 6: Steps for plugging PiCO QL in an application.



information. Then it generates virtual table definitions and C++ functions to address SQL queries. Queries target virtual tables, but internally they are resolved by executing the generated C++ code, which operates on the C++ application code data structures. The generative programming component of PiCO QL was implemented in Ruby (Flanagan and Matsumoto, 2008).

4.2. Virtual table implementation

PiCO QL implements an SQLite virtual table module to provide a relational interface to C++ data structures. Specifically, the implementation includes a number of callback functions that specify the PiCO QL virtual table's behavior: create, destroy, connect, disconnect, open, close, filter, column, plan, advance_cursor and eof. The SQLite query engine calls these functions when performing a query on a PiCO QL virtual table. Of all callback functions, *filter* and *column* are specific to a PiCO QL virtual table and are generated according to the relational specification of the C++ data structure that the virtual table represents (Listing 12). A hook in the query planner (*plan* callback function) ensures that the constraint referencing the base column has the highest priority in the constraint set for the VT_n and, therefore, the

instantiation will happen prior to evaluating any real constraints.

Listing 12: Software structure of relational representation

```
int Account_VT_search(...) {
    ...
    switch(col) {
    case 0:
        for (int i = 0; i < accounts.size(); i++) {
            if (compare(accounts[i].description, operator, rhs)
                add_to_result_set ();
        }
        break;
    case ...
    ...
```

In query processing, PiCO QL and SQLite share responsibilities. PiCO QL controls query planning through an implemented callback function and carries out constraint and data management for each virtual table. SQLite performs high level query evaluation and optimization (Owens, 2006, p. 360).

Having virtual table instantiations come into existence is not hard or computationally intensive. A virtual table can be thought of as a concept whose rules are defined in the DSL. Data structures that adhere to the concept's rules use its representation and instantiate the virtual table. In effect, a virtual table is a structure that the query engine uses when the virtual table is referenced in a query. Multiple instantiations use the same structure and the structure stores a reference to its current instantiation; a new virtual table instantiation has the cost of a pointer dereference.

4.3. Query interface

A user interface is required in order to issue queries and view the results. For this we adopted SWILL, a library that adds a web interface to C/C++ programs. Each web page in SWILL is a C function that blends HTML and C/C++ application code to present useful information about an application. For a query interface three such functions were used, one to input queries, one to output query results, and one to display errors. Using SWILL as a bridge the user interface can interact with SQLite easily through SQLite's C API. Queries are interpreted by the SQLite engine, which in turn calls the virtual table implementation's callback functions (section 4.2).

5. Evaluation of PiCO QL's query mechanism

PiCO QL's evaluation follows the Goal-Question-Metrics approach (Basili et al., 1994). The *goal* is to show that the current work offers an improved solution compared to alternatives. Two *questions* will drive the answers required to achieve the goal, specifically:

1. How does the current work compare to alternatives in terms of expressiveness?
2. How does the current work compare to alternatives in terms of temporal and spatial efficiency?

Finally, the selected *metrics* consist of:

- lines of code for measuring expressiveness
- CPU time in seconds for measuring temporal efficiency.
- storage space for measuring spatial efficiency, both for storing the code on disk and for storing the data in memory during execution.

Table 1: Projects used in evaluation

Project	URL	**Size** (KLOC)
Stellarium	`http://www.stellarium.org/`	888
QLandKarte	`http://www.qlandkarte.org/`	39
CScout	`http://www.spinellis.gr/cscout/`	18

PiCO QL has been evaluated on three large C++ projects, Stellarium, QLandKarte, and CScout (Spinellis, 2010) (see Table 1). In the case studies we compare PiCO QL queries to equivalent queries expressed using C++ constructs with respect to the described metrics. In addition, CScout can dump the relationships of an entire workspace in the form of an SQL script. This can then be uploaded into a relational database for further querying and processing. Hence for CScout we also carry out the measurements in a MySQL database with a default configuration and enabled indexes. The PiCO QL evaluation queries for the case studies, the PiCO QL DSL artifacts, the C++ queries and the MySQL queries are available online[2],[3],[4] except for

[2] `https://github.com/mfragkoulis/PiCO_QL/tree/master/examples/Cscout`
[3] `https://github.com/mfragkoulis/PiCO_QL/tree/master/examples/QLandKarte`

CScout's C++ queries, which are embedded in CScout's plain query facility. We list the PiCO QL DSL for the Stellarium case study in Listing 13.

Listing 13: Stellarium DSL virtual table descriptions

```
CREATE STRUCT VIEW Meteor (
    velocity  DOUBLE FROM velocity,
    magnitude FLOAT FROM mag,
    observDistance DOUBLE FROM xydistance)$

CREATE VIRTUAL TABLE Meteor
USING STRUCT VIEW Meteor
WITH REGISTERED C NAME active
WITH REGISTERED C TYPE vector<Meteor*>$

CREATE STRUCT VIEW Planet (
    name STRING FROM data()->getNameI18n().toStdString(),
    hasAtmosphere BOOL FROM data()->hasAtmosphere(),
    distance DOUBLE FROM data()->getDistance(),
    FOREIGN KEY(satellites_id) FROM data()->satellites.toStdList()
                    REFERENCES SatellitePlanet)$

CREATE VIRTUAL TABLE Planet
USING STRUCT VIEW Planet
WITH REGISTERED C NAME allPlanets
WITH REGISTERED C TYPE list<PlanetP>$

CREATE VIRTUAL TABLE SatellitePlanet
USING STRUCT VIEW Planet
WITH REGISTERED C TYPE list<QSharedPointer<Planet> >*$
```

5.1. Case study 1: Stellarium

Stellarium is an open source virtual real time observatory of stellar objects. It hosts and presents useful information about stellar objects through its user-driven GUI. SQL queries in Stellarium's use case concern planets and meteors typically stored in C++ containers. Each container contains approximately 100 elements. Measurements of PiCO QL and C++ queries took place at the same machine[5] under identical (mostly idle) load. Each measurement represents the minimum value obtained over 100 runs.

[4] `https://github.com/mfragkoulis/PiCO_QL/tree/master/examples/Stellarium`

[5] Mac OS X 10.6.8, 2.4 GHz intel Core 2 Duo, 2 GB 667 MHz DDR2 SDRAM

5.2. Case study 2: QLandKarte

QLandKarte is an open source GIS application that displays GPS data on a variety of maps. SQL queries in QLandKarte's use case concern waypoints of 4 thousand data elements. Measurements of PiCO QL and C++ queries took place at the same machine under (mostly idle) identical load. Each measurement represents the minimum value obtained over 10 runs.

5.3. Case study 3: CScout

CScout is a source code analyzer and refactoring browser for collections of C programs. It can process workspaces of multiple projects (we define a project as a collection of C source files that are linked together) mapping the complexity introduced by the C preprocessor back into the original C source code files. In this case study we use PiCO QL, C++, and MySQL to perform sophisticated queries on the extracted identifiers, files, functions, and function-like macros of the Linux kernel. CScout containers store 1.1 million identifiers and 89 thousand functions and function-like macros. All measurements took place at the same machine[6] under (mostly idle) identical load. Each measurement represents the minimum value obtained over 3 runs.

5.4. Evaluation measurements presentation

Evaluation measurements consist of LOC, CPU execution time, query memory use, and marginal query code size measurements.

5.4.1. LOC measurements

The code query size is listed in Table 2. The cost of supplying a relational representation for querying the application's data structures has not been accounted in PiCO QL measurements. This is an one-off cost amortized over use. CScout contains three interfaces for performing queries on processed identifiers, files, and functions. The measurements amount to the C++ query facility code used for calculating each of the three evaluation queries. Query facility code does not include the generic code base shared by all three interfaces (203 LOC) and the presentation layer. Query listings are available online.[7]

[6]Linux 2.6.32-5-amd64, 8 Dual Core AMD Opteron(tm) Processor 880 CPUs, 16 GB RAM

[7]All query listings are included under `https://github.com/mfragkoulis/PiCO_QL/tree/master/examples/`

Table 2: LOC measurements

Case study	Query Listing	PiCO QL	C++	MySQL
Stellarium	Stellarium/get_meteors.sql	7	31	N/A
	Stellarium/get_planets1.sql	8	33	N/A
	Stellarium/get_planets2.sql	6	40	N/A
QLandKarte	QLandKarte/get_points1.sql	6	76	N/A
	QLandKarte/get_points2.sql	7	153	N/A
	QLandKarte/get_points3.sql	9	38	N/A
CScout	CScout/get_identifiers.sql	10	425	17
	CScout/get_files.sql	8	425	13
	CScout/get_functions.sql	5	433	7

5.4.2. CPU execution time measurements

In calculating CPU execution time (Table 3) we use the built-in C time library in PiCO QL and C++ queries and the built-in MySQL query report in MySQL queries. MySQL time measurements exclude data import time.

The C++ queries in Stellarium and QLandKarte case studies were implemented without putting effort to achieve optimization. Specifically, we managed groupings in C++ using an associative container, such as a `map`. An additional group by term requires an additional container embedded in the first. In fact a `multimap` is convenient for accommodating a second group by term since it provides the opportunity to group values of the second term for which the first term has the same value. Both containers were heavily used in evaluation queries. Algorithmic support for container operations is limited, but most of the time it is sufficient to express the equivalent of a *having* clause. In some evaluation queries, custom algorithms were used.

Table 3: CPU execution time

Case study	Query Listing	PiCO QL	C++	MySQL
Stellarium	Stellarium/get_meteors.sql	749μs	163μs	N/A
	Stellarium/get_planets1.sql	1538μs	774μs	N/A
	Stellarium/get_planets2.sql	1020μs	739μs	N/A
QLandKarte	QLandKarte/get_points1.sql	184ms	8637ms	N/A
	QLandKarte/get_points2.sql	161ms	9385ms	N/A
	QLandKarte/get_points3.sql	90ms	8564ms	N/A
CScout	CScout/get_identifiers.sql	2190ms	1070ms	123.30s
	CScout/get_files.sql	2220ms	1010ms	580ms
	CScout/get_functions.sql	260ms	300ms	580ms

5.4.3. Query memory use measurements

For calculating memory space during query processing (Table 4) we used *ltrace* for PiCO QL and C++ queries in CScout under Linux, maximum resident set size reported by the GNU time utility for MySQL queries, and maximum resident set size reported by GNU *rusage* in Stellarium and QLandKarte tested under Mac OS X. Each MySQL query run took place on a freshly started database server without cleaning the machine's buffer cache; each measurement represents the observed peak resident set size at the database client. Similarly, to get a reliable measurement of the peak resident set size from GNU time and GNU resource usage we restarted the application after each run.

Table 4: Query memory use

Case study	**Query Listing**	PiCO QL	C++	MySQL
Stellarium	Stellarium/get_meteors.sql	61kB	4kB	N/A
	Stellarium/get_planets1.sql	59kB	12kB	N/A
	Stellarium/get_planets2.sql	41kB	49kB	N/A
QLandKarte	QLandKarte/get_points1.sql	971kB	2683kB	N/A
	QLandKarte/get_points2.sql	967kB	2826kB	N/A
	QLandKarte/get_points3.sql	266kB	2572kB	N/A
CScout	CScout/get_identifiers.sql	12kB	7kB	320kB
	CScout/get_files.sql	102kB	6kB	364kB
	CScout/get_functions.sql	6161kB	6kB	7155kB

5.4.4. Marginal query code size measurements

Marginal query code size (Table 5) was calculated by compiling the studied applications with and without query code.

5.5. Evaluation outcomes

PiCO QL reduces the amount of code for expressing an SQL query in C++ and even seems to have enhanced expressive power compared to SQL. Each line in a PiCO QL query corresponds to six lines of C++ code on average. GROUP BY clauses cause a significant fraction of this expressiveness gap.

The difference, in favour of PiCO QL, that we observe betweeen PiCO QL and MySQL queries is explained by the reduced normalization required in PiCO QL. Specifically, we have chosen to model 1:1 associations in the same virtual table (recall section 3.1.1 for further explanation on PiCO QL

Table 5: Marginal query code size

Case study	Query Listing	PiCO QL	C++	MySQL
Stellarium	Stellarium/get_meteors.sql	234kB	47kB	N/A
	Stellarium/get_planets1.sql	234kB	61kB	N/A
	Stellarium/get_planets2.sql	234kB	34kB	N/A
QLandKarte	QLandKarte/get_points1.sql	201kB	19kB	N/A
	QLandKarte/get_points2.sql	201kB	36kB	N/A
	QLandKarte/get_points3.sql	201kB	32kB	N/A
CScout	CScout/get_identifiers.sql	941kB	N/A	N/A
	CScout/get_files.sql	941kB	N/A	N/A
	CScout/get_functions.sql	941kB	N/A	N/A

modelling), whereas in a typical relational schema there would be a table for each entity participating in the association and a primary key / foreign key relationship instance to accommodate the association. As a result, each PiCO QL evaluation query saves two joins or four LOC on average.

CPU execution time measurements show that speed mainly depends on data sizes, optimization effort and specific query operations. Querying using C++ programming constructs is more efficient for small data sizes (Stellarium case study). Managing groupings in queries favors PiCO QL over C++ up to a factor of 58 for moderate data sizes (QLandKarte case study) and for naive C++ query implementations, that is without putting effort to achieve optimization.

Regarding the CScout case study, MySQL with indexes enabled is very efficient but for some operations object-oriented query processing is still faster (`CScout/get_identifiers.sql`). For simple conditional operations C++ queries are twice as fast as PiCO QL in two out of three cases. In the remaining case, PiCO QL slightly outperforms the C++ query implementations and is twice as fast as MySQL. Since the corresponding query returned a large result set (> 20000 records), a possible explanation is that PiCO QL performs better in result set presentation.

Query memory use measurements show that MySQL queries consume most memory and C++ queries consume minimum space for simple cases. PiCO QL stands in between having a modest memory footprint.

Marginal query code size is considerably larger for PiCO QL compared to each C++ query but this is actually the space for the library as a whole. The cost is amortized over an arbitrary number of queries.

6. Discussion

Table 6: Synopsis of query languages

Query language	*sbql4j*	MemSQL	QBD	HLQLs	Daytona	PiCO QL
Computational power	⊂SQL	SQL	⊂SQL	⊃SQL	⊃SQL	SQL
OO support	✓	✓	✓	x	x	✓
Queries to program data	x	x	✓	x	x	✓
Queries to file data	x	x	x	✓	✓	x
Queries to database data	x	✓	x	✓/x	✓	x
Main memory operation	✓	✓	✓	x	✓	✓
Parallel operation	x	✓	x	✓	✓	x

PiCO QL presents distinct objectives and characteristics compared to the related pieces of work described in section 2.

PiCO QL does not belong to the class of software known as object-relational mapping software (Bauer and King, 2006; Melnik et al., 2007); it utilizes an object-relational mapping technique to provide a relational interface to the object-oriented model. In PiCO QL in addition to classes object containers and shared objects can be tables. This results from the way a relational interface is overlaid on OO data (see section 3), guaranteeing a clear relational representation of OO data structures. Accessibility, inheritance, and polymorphism follow the programming language rules.

As queries in PiCO QL take the form of standard SQL, an advantage of the approach is the small learning effort required for the query language. PiCO QL utilises path expressions in query execution internally to anchor a C++ class member to a virtual table column. By providing an anchor for each column, visible C++ class attributes or methods can be referenced by an SQL query. By contrast, object-oriented query languages use path expressions directly in queries.

Main memory DBMSs with an external query interface share some conceptual similarities to the current work. The most important is leveraging

pointers for cross-referencing objects. The main difference is that PiCO QL performs querying in place, on the program data space, contrary to DBMSs, which copy and manage data in own data structures. MemSQL targets C++, but its implementation details are unavailable. Daytona is a proprietary system that supports C applications. Daytona and PiCO QL have in common a code-generating architecture, but Daytona uses a compiled procedure for query processing (as described in section 2.2.1) compared to the interpreted approach adopted by PiCO QL. In computation terms, Daytona's query language Cymbal includes ANSI 89 SQL, and in addition a procedural dialect with a first-order logic subset, a sublanguage for set/list formers and a sublanguage for database record description. Although ANSI 92 SQL, which PiCO QL supports, includes ANSI 89 SQL, Cymbal is computationally stronger.

PiCO QL and HLQLs exhibit a number of common traits. They use a data schema only as a means of representation and do not require a data import. Both PiCO QL and HLQLs provide read-only ad-hoc data analysis, and, thus, avoid the complexity of complete data management. In addition, they provide a user interface for queries and results. Finally, PiCO QL and HLQLs except JAQL provide an SQL flavor of non-relational non-database data. SCOPE and HIVEQL even represent data as relational tables.

There are, however, important differences. PiCO QL targets data structured as C++ objects in memory using a relational interface. HLQLs use a map-reduce, or another highly-parallel paradigm, and define their own type system. PiCO QL uses a typical relational type system. HLQLs are capable of offline analysis of files while PiCO QL is used for online analysis of C++ application data. HLQLs are widely used in production against big data challenges. HLQLs allow customization of output and storage. On the other hand, in PiCO QL it is possible to define standard relational (non-materialized) views in the DSL following the standard SQL syntax. Views become part of the virtual table schema as per SQLite's support. HLQLs allow user defined functions to take part in queries. In PiCO QL this is feasible by writing a function or module and loading it to the SQLite query engine. Then it becomes available for use in SQL queries.

All in all, PiCO QL does not compete against HLQLs. It mainly provides query computation over in-memory C++ datasets à la SQL, not massively parallel query execution in a map-reduce fashion, or other, over datasets located in distributed file systems.

Table 6 outlines PiCO QL's contribution by summarizing the most important characteristics of the afore-mentioned related categories of query

languages and of this work.

7. Conclusions

We presented the design and implementation of an approach for mapping the object-oriented model to a relational interface. The approach allows SQL queries to execute on object-oriented data structures through their relational representation. The implementation, PiCO QL, delivers a usable SQL interface for interactive, ad-hoc queries to C++ program objects. Its evaluation shows query expressiveness, often enhanced query speed and low memory footprint. For applications that only require a DBMS's query facilities, a fully-fledged DBMS is superfluous. Introducing it would mean adding an intrusive dependency, extra overhead, and writing boilerplate code for interacting with the application, which would litter application code. PiCO QL is valuable to those applications by offering a lightweight SQL interface for querying the application's data structures. PiCO QL could be advantageous in situations that require managing software state, like application server resource monitoring, stream processing and scientific computing. Future work will target performance optimizations and extending support of data structures and algorithms.

Code availability

The full source code of PiCO QL, wiki, and examples can be found in `https://github.com/mfragkoulis/PiCO_QL`. The code is available under the Apache license.

Acknowledgements

This research has been co-financed by Foundation Propondis and by the European Union (European Social Fund — ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) — Research Funding Program: Thalis — Athens University of Economics and Business — Software Engineering Research Platform.

Adamus, R., Habela, R., Kaczmarski, K., Lentner, M., Stencel, K., Subieta, K., 2008. Stack-based architecture and stack-based query language. In: ICOODB. pp. 77–96.

Basili, V. R., Caldiera, G., Rombach, H. D., 1994. The Goal Question Metric approach. In: Encyclopedia of Software Engineering. Vol. 1. Wiley, pp. 528–532.

Bauer, C., King, G., Nov. 2006. Java Persistence with Hibernate, revised Edition. Manning Publications.

Beyer, K. S., Ercegovac, V., Krishnamurthy, R., Raghavan, S., Rao, J., Reiss, F., Shekita, E. J., Simmen, D. E., Tata, S., Vaithyanathan, S., Zhu, H., 2009. Towards a scalable enterprise content analytics platform. IEEE Data Eng. Bull. 32 (1), 28–35.

Cattell, R. G. G., Barry, D. K. (Eds.), 2000. The object data standard: ODMG 3.0. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Chaiken, R., Jenkins, B., Larson, P. A., Ramsey, B., Shakib, D., Weaver, S., Zhou, J., Aug. 2008. SCOPE: Easy and efficient parallel processing of massive data sets. Proc. VLDB Endow. 1 (2), 1265–1276.

Czarnecki, K., Foster, J. N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J. F., 2009. Bidirectional transformations: A cross-discipline perspective. In: Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations. ICMT '09. Springer-Verlag, Berlin, Heidelberg, pp. 260–283.

Dean, J., Ghemawat, S., Jan. 2008. Mapreduce: simplified data processing on large clusters. Commun. ACM 51 (1), 107–113.

Flanagan, D., Matsumoto, Y., 2008. The Ruby programming language, 1st Edition. O'Reilly.

Frohn, J., Lausen, G., Uphoff, H., 1994. Access to objects by path expressions and rules. In: Proceedings of the 20th International Conference on Very Large Data Bases. VLDB '94. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 273–284.

Goldsmith, S. F., O'Callahan, R., Aiken, A., 2005. Relational queries over program traces. In: Proceedings of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA '05. ACM, New York, NY, USA, pp. 385–402.

Greer, R., 1999. Daytona and the fourth-generation language Cymbal. In: Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data. SIGMOD '99. ACM, New York, NY, USA, pp. 525–526.

Hawkins, P., Aiken, A., Fisher, K., Rinard, M., Sagiv, M., 2011. Data representation synthesis. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '11. ACM, New York, NY, USA, pp. 38–49.

Keith, M., Schincariol, M., 2009. Pro JPA 2: Mastering the Java Persistence API, 1st Edition. Apress, Berkely, CA, USA.

Lampoudi, S., Beazley, D. M., 2002. SWILL: A simple embedded web server library. In: Demetriou, C. G. (Ed.), Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA. USENIX, pp. 19–27.

Lehman, T. J., Carey, M. J., 1986. Query processing in main memory database management systems. In: Proceedings of the 1986 ACM SIGMOD international conference on Management of data. SIGMOD '86. ACM, New York, NY, USA, pp. 239–250.

Lencevicius, R., Hölzle, U., Singh, A. K., Jan. 2003. Dynamic query-based debugging of object-oriented programs. Automated Software Eng. 10 (1), 39–74.

Meijer, E., Oct. 2011. The world according to LINQ. Commun. ACM 54 (10), 45–51.

Meijer, E., Beckman, B., Bierman, G., 2006. LINQ: Reconciling object, relations and XML in the .NET framework. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data. SIGMOD '06. ACM, New York, NY, USA, pp. 706–706.

Melnik, S., Adya, A., Bernstein, P. A., 2007. Compiling mappings to bridge applications and databases. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data. SIGMOD '07. ACM, New York, NY, USA, pp. 461–472.

Melnik, S., Gubarev, A., Long, J. J., Romer, G., Shivakumar, S., Tolton, M., Vassilakis, T., Sep. 2010. Dremel: Interactive analysis of web-scale datasets. Proc. VLDB Endow. 3 (1-2), 330–339.

MemSQL, 2013. The MemSQL database. Available online `http://memsql.com/` Current March 2013.

Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A., 2008. Pig Latin: A not-so-foreign language for data processing. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. SIGMOD '08. ACM, New York, NY, USA, pp. 1099–1110.

Owens, M., 2006. The Definitive Guide to SQLite (Definitive Guide). Apress, Berkely, CA, USA.

Reese, G., 2000. Database Programming with JDBC and Java, 2nd Edition. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Sanders, R. E., 1998. ODBC 3.5 Developer's Guide. McGraw-Hill Professional.

Schwartz, J. T., Dewar, R. B., Schonberg, E., Dubinsky, E., 1986. Programming with sets; an introduction to SETL. Springer-Verlag New York, Inc., New York, NY, USA.

Spiewak, D., Zhao, T., 2010. ScalaQL: Language-integrated database queries for Scala. In: Proceedings of the Second International Conference on Software Language Engineering. SLE '09. Springer-Verlag, Berlin, Heidelberg, pp. 154–163.

Spinellis, D., Apr. 2010. CScout: A refactoring browser for C. Sci. Comput. Program. 75 (4), 216–231.

Stewart, R. J., Trinder, P. W., Loidl, H. W., 2011. Comparing high level mapreduce query languages. In: Proceedings of the 9th International Conference on Advanced Parallel Processing Technologies. APPT '11. Springer-Verlag, Berlin, Heidelberg, pp. 58–72.

Stonebraker, M., Cetintemel, U., 2005. "one size fits all": An idea whose time has come and gone. In: Proceedings of the 21st International Conference on Data Engineering. ICDE '05. IEEE Computer Society, Washington, DC, USA, pp. 2–11.

The SQLite team, 2013. The virtual table mechanism of SQLite. Available online `http://www.sqlite.org/vtab.html` Current March 2013.

Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Antony, S., Liu, H., Murthy, R., Mar. 2010. Hive - a petabyte scale data warehouse using Hadoop. In: ICDE '10: Proceedings of the 26th International Conference on Data Engineering. IEEE, pp. 996–1005.

Wcislo, E., Habela, P., Subieta, K., 2011. A Java-integrated object oriented query language. In: Abd Manaf, A., Zeki, A., Zamani, M., Chuprat, S., El-Qawasmeh, E. (Eds.), Informatics Engineering and Information Science. Vol. 251 of Communications in Computer and Information Science. Springer Berlin Heidelberg, pp. 589–603, 10.1007/978-3-642-25327-0_50.

Willis, D., Pearce, D. J., Noble, J., 2006. Efficient object querying for Java. In: Proceedings of the 20th European Conference on Object-Oriented Programming. ECOOP '06. Springer-Verlag, Berlin, Heidelberg, pp. 28–49.

Marios Fragkoulis is a graduate researcher at the Athens University of Economics and Business (AUEB), Department of Management Science and Technology. He holds a BSc in Management Science and Technology from the Athens University of Economics and Business (Distinction) and a MSc in Computing Science from Imperial College London (Distinction).

Diomidis Spinellis is a Professor in the Department of Management Science and Technology at the Athens University of Economics and Business, Greece. He is the author of two award-winning books, Code Reading and Code Quality: The Open Source Perspective. He is a member of the IEEE Software editorial board, authoring the regular Tools of the Trade column. Dr. Spinellis holds an MEng in Software Engineering and a PhD in Computer Science, both from Imperial College London and is senior member of the ACM and the IEEE.

Panos Louridas is a senior researcher at Department of Management Science and Technology of the Athens University of Economics and Business and a program manager at

the Greek Research and Education Network (GRNET), responsible for cloud computing projects. He has published in all aspects of software engineering while remaining a practising developer. He holds a Diploma in Informatics from the University of Athens and a MSc by Research and a PhD from the University of Manchester. He is a member of the ACM, the IEEE, USENIX, and AAAS.