Annales UMCS Informatica AI 7 (2007) 143-151

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI

http://www.annales.umcs.lublin.pl/

PySBQL – Python-like query language constructed using stack base approach

Marta Rogińska, Piotr Wiśniewski*

Faculty of Mathematics and Computer Science, Nicolaus Copernicus University, Toruń, Poland

Abstract

PySBQL (Python-like Stack Based Query Language) is a full scale programming and query language. Its syntax is based upon the Python's syntax, which makes PySBQL highly readable and easy to use. Contrary to the classical approach in query languages, semantics is defined using a common structure for programming languages – the Environment Stack (ENVS). As a query language it is similar to SBQL proposed by Subieta [1,2]. The PySBQL language is implemented in Monad – Object Oriented Database Management System.

1. Introduction

The same years the tendency towards objectivity has been observed. The relational data model does not fulfill the needs of the databases users. A some alternatives for the SQL query language has been searched for. Since the first databases and the first query languages were created there has been a need to combine them with a programming language. Many attempts to embed a query language inside a programming language like C or Java showed failure. The proper solution should combine the attributes of a query language and a programming language. The most important aspects would be semantic precision, simplicity, and code readability.

The classical design process of database applications is burdensome. One problem is typological incompatibility, the other one is the difference between the early binding of the programming language and the late binding of the query language. Those problems are known as impedance mismatch. The procedural query languages (PL/SQL, Transact SQL, etc.) are not a good solution, because they are not fully featured programming languages. Therefore a query language with a functionality of the programming language is needed. Such language should have precise semantics with no "reefs". It should result in a simple,

*Corresponding author: *e-mail address*: pikonrad@mat.umk.pl

readable code. The need for declaring variables and type pre-assigning can be considered redundant. The only necessity is that a value stored within a variable should be of a precisely determined type.

In this paper we describe PySBQL – a language which is a synthesis of Python language and SBQL. The SBQL is a prototype language characterized by the stack approach, designed by K.Subieta and coworkers [1,2]. PySBQL's syntax is mostly based on Python's syntax enlarged by the query operators construction. Semantics, similarly to SBQL, is based on the environment stack and the query result stack. This leads to a compact language with no redundancy in syntax, and no distinction in grammar between the database and the programming language operators. Our goal in PySBQL design was to make it as similar to Python as possible in order to make it easy to master (especially for Python programmers). This will allow import to the new language Python's libraries and programs with only minimal modifications. Unfortunately, a number of operators has been changed so full compatibility has not been achieved.

2. Impedance mismatch

The impedance mismatch is a well known problem when dealing with mapping data between the database systems and the programming languages (mostly Java nowadays). It stems from differences in syntax, namespaces, data abstraction levels, scope rules and typology differences, to name a few.

Addition of new features to existing languages does not solve those problems, as it only decreases the discrepancies. A common example is JDBC library for Java. It allows for certain type control using the methods like `setBlob()` or `getFloat()` but it still needs to embed SQL queries in the Java methods which belong to a completely different style of grammar. Even slightly complex queries lack readability and the code is difficult to maintain.

All those problems could be solved by means of a new tool, namely a language that allows for seamless blending of query and programming language operators. It should allow for traditional programming language expressions like literals, variables or operators, as well as should provide tools for querying and manipulating databases. This tool could be used to communicate with a DBMS, and to create full scale applications. In 1993, Kazimierz Subieta proposed a concept of the Stack Based Query Language (SBQL) [1], which fulfills the aforementioned goals.

3. Data model

Subieta proposed a set of store models that could be used in the Object DBMS. The basic model is called M0 (first defined in [1] and called there the

Abstract Data Model). Within it there are three types of objects: atomic, pointer and complex. Each of them has a unique, internal identifier, external name and value. For the atomic object the value is of a simple type, like a number or a string. Pointer object's value is the identifier of another object. There can be no pointers without reference to a target object from the database. For the complex object the value is a set of objects with no limitations to their quantity or level of hierarchy. Objects of the same name may be of different types or may contain different amount of sub-objects. In the model M0 an object store is a pair (O,R) where O is a set of objects and R is a set of the top-most objects (roots). A sample database is presented in example 3.1.

Example 3.1. Example of a simple database in a graphical form:

```
<i1, empl,{
   <i2, fname, “John”>
   <i3, sname, “Smith”>
   <i4, dept, i20>
   <i5, salary, 2000>
   }>
<i6, empl,{
   <i7, fname, “Bob”>
   <i8, sname, “Gordon”>
   <i9, dept, i20>
   <i10, salary, 2300>
   }>
<i11, empl,{
   <i13, sname, “Watson”>
   <i14, dept, i30>
   }>
<i20, dept, {
   <i21, name, “IT”>
   <i22, employee, i1>
   <i23, employee, i2>
   <i24, boss, i6>
   }>
<i30, dept, {
   <i31, name, “administration”>
   <i32, employee, i11>
   <i33, boss, i11>
   }>
R=[i1,i6,i11,i20,i30]
```

The next model proposed is M1. It is basically the M0 model extended by the use of classes and inheritance. It is obtained by an addition of a set of classes' identifiers C and 2 relations: CC that determines inheritance among classes, and OC that determines the membership of objects in classes. Classes are stored as complex objects, yet their identifiers do not belong to O. The CC relation cannot contain cycles. Classes are stored as complex objects as shown in example 3.2.

Example 3.2.

```
<i100, class,{
   <i101, classname, "empl">
   ....
   other information of class empl
   ....
   }>
<i200, class,{
   <i201, classname, "dept">
   ....
   other information of class dept
   ....
   }>
OC=[(i1,i100),(i6,i100),(i11,i100),(i20,i200),(i30,i200)]
```

This model has been successfully adapted in a few experimental object databases management systems.

4. PySBQL

In this paper we present PySBQL – a tool for data manipulation and applications design. It is a language that combines the concept of SBQL and features of the Python programming language [3]. PySBQL deals with expressions and queries in the same manner, thus in the following sections the words *expression*, *query* and *statement* will be used interchangeably. The basis of PySBQL lies in the M1 data model. We impose no limitations on complex objects. The only constraints are derived from the object's class definition. That is to say a complex object may contain many instances of the same class of objects (e.g. Job) or may contain no sub-objects.

In PySBQL, the basic queries are literals (numbers, strings) or names. Each more complex query is constructed of sub-queries. It is a basically common approach for programming languages. Similarly to Python, syntax is based upon the indentations. This results in a highly readable code. If an imperative construction like `for` is to be followed by a block of statements, then the

programmer needs to increase the indentation. The end of the block is denoted by the decrease of the indentation level to one of the previous levels. A sample code is presented in an example 4.1.

Example 4.1. Example of PySBQL's code:

```
for p in (dept where name == „IT").employee:
    p.salary *= 1.15
    if (p.salary > p.dept.boss.salary):
        p.salary, p.dept.boss.salary=p.dept.boss.salary, p.salary
    print p.(fname, sname, salary)
```

There are six kinds of query results: literals, references, tuples, lists, dictionaries and binders. Binder is a pair (name, result), usually written: name(result). All the results may be combined and nested in a fully orthogonal way, but they are not objects – they have no identifiers and may have no names. Function parameters belong to the same domain as results, therefore it is possible to pass a query as a parameter. Like in Python, a programmer does not have to place constraints on the number of function parameters. A sample function definition is presented in example 4.2. It is also worth mentioning that a function may return a reference to another function as a result.

Example 4.2. Example of a function's definition and its usage:

```
def averegeWage(*wages):
    i,sum=0,0.0
    for w in wages:
        i+=1
        sum+=w
    else:
        print „No wages given"
    print sum/i
averegeWage((employee where dept.name=="Sales").salary)
```

4.1. Evaluation

The evaluation is based upon an Environment Stack (ENVS). It is divided into sections, called environments, filled in with binders. As in [2], one of the bottom sections is a database section containing binders to root objects. The difference between the stack used in PySBQL and the classical ENVS is that a section of PySBQL's ENVS may contain more than one binder with a given name. During the evaluation of a statement, the occurring names are bound using the ENVS. Example 4.1.1 presents evaluation of a simple query.

Example 4.1.1.

Let us discuss the simplest query:

```
empl
```

It is recognized as a name, and it is searched for on the ENVS's sections. The search is stopped when a section with at least one binder with a name `empl` is found. The result is a tuple containing the content of matching binders from this section. For the database example 3.1 there are three binders to the name `empl` in the database section, so the result is a tuple `(i1,i6,i11)`

Variables are simply binders placed in the proper sections. Their type is not associated with a name but it is derived from value, like implemented in Python. Although the variables are not declared, our language has a strong type control. PySBQL is an interpreted language therefore the control is also dynamic. This way the language allows for more flexible approach to the data being stored and manipulated, but still it gives a certain safety when manipulating data of unknown type. This problem is rarely present in programming languages, but it is very common in databases and query languages. In the traditional query processing in a host language the type control, if present at all, is difficult and troublesome.

4.2. Operators

The operators in PySBQL are divided into 3 main groups: algebraic, non-algebraic and imperative operators. The division is mainly based upon the way of evaluation. Non-algebraic operators are evaluated with the help of ENVS, and they cause new stack sections to appear. Having a query *q1 <NAop> q2*, it is usually evaluated by application of operator to the sub-query *q2* in the context of the result of sub-query *q1*. Among such operators are . (dot), `where`, `order by` etc. Example 4.2.1 presents an evaluation process for the non-algebraic operator `where`.

Example 4.2.1.

```
empl where sname == "Smith"
```

Let us analyze this query in the context of example 3.1. The operator `where` will firstly evaluate the left sub-query `empl`. The result will be a tuple `(i1,i6,i11)`. Then for every object `e=i1,i6,i11` the operator `where` will build a section in the ENVS with binders to elements inside of `e` (note, that `e` is a complex object). Starting with `i1`, within a new section a binder `sname(i3)` is found. After evaluating `sname=="Smith"` the operator `where` will delete this section. Then it will create a new section for `i6`, and after checking the condition – again for `i11`. In the first case the condition `sname == "Smith"`

will be evaluated into the True value, in the rest of the cases the value will be false. As a consequence the operator `where` will return `i1` as a result.

It should be noted that after the evaluation of the non-algebraic operator, ENVS returns to its previous content. The algebraic operators, that do not use the Environment Stack directly, are evaluated by the application of the operator to the results of sub-queries. Within this group of operators there are all arithmetical and logical operators. Imperative operators evaluate on the ENVS and modify the database. After evaluation of an imperative operator, the ENVS could be modified. As an example `a=7` may insert a new binder into the top section (if `a` binder a does not exist previously). Among those operators there are object creation, insertion, assignment, deletion, etc. They are necessary in order to work with a database.

Another interesting group contains iterators like `while`, `if`, and `for`. Their structure and grammar are taken from the Python language, but their semantics is slightly changed. The main difference is visible within the for operator. To allow queries such as

```
for n, surn in empl.(fname,sname):
    print n, surn
```

Some changes in the way the `for` operator treats missing values had to performed. In our example one of the `empl`s does not have a `fname` sub-object. In Python's `for` construction an error would be raised. In PySBQL the `None` value is assigned to the variable. There are other slight differences regarding the grammar of those operators and the approach to sub-queries as the part of conditions. Iterators open a new section on the Environment Stack as they are usually followed by blocks of operators.

The operators like . (dot), `as`, + or the comparison operators have been modified to work with objects from a database. The dot operator may now be followed by a list of arguments, that are, in fact, sub-queries. Example 4.2.2 shows the dot operator within a context that would be invalid for Python.

Example 4.2.2

```
print empl.("Mr", fname, sname, "earns", salary)
```

The operator *as* in PySBQL creates synonyms for an object within a certain query, + operator performs operation of adding elements of one tuple to another (this is present in some of the implementations of Python, but is not present in the standard). PySBQL also has few keywords that are not present in either Python nor SBQL. Most important of them are <- and `this`. <- is used to insert an object into another one, and `this` keyword is used in non-algebraic operator

to refer to the object nested on the top of the ENVS. Its use is presented in the following example:

```
print (empl where sname==“Smith”).(this, dept.boss)
```

The combination of imperative, algebraic and non-algebraic operators together with the Environment Stack and iterators form a full functionality, that a query and a programming language should possess.

The PySBQL classes are considered complex objects which may contain other objects. Therefore their structure may change during runtime. Unlike many programming languages PySBQL allows for multiple inheritance. This is dictated by the need to represent real life correspondences.

The source of some “semantic reefs” in SQL is a NULL value. There are no NULL values in PySBQL – missing information is simply not recorded, although there exists a literal type: `NoneType` containing one element: `None`. It is taken directly form the Python language.

5. Monad project

Monad is an object oriented database management system based on the PySBQL language. Monad is a fully scalable programing platform. We can use it as an interpreter to simple scripts, simple database applications, distributed database applications with many aspects of distribution: classical client – server, peer to peer net or other grid architecture. The Communication between instances is based on a SOAP protocol. The PySBQL language with the Monad system is dedicated to the programmers familiar with the Python language. Monad will not support JDBC compliant API, because it does not support any form of SQL. The JDBC driver is under construction, and it will be an API with some kind of native queries. In the future Monad will integrate with WebServises using OGSA-DAI technology to enable its integration with Globus and other grid technologies. Another feature which will be implemented soon is updatable views. They are known to be a difficult problem for the relational and object-relational database management systems. The work by H. Kozankiewicz [4,5] showed that updatable views are less problematic for query languages based on the Stack Approach. Some usable features can be implemented as a part of system library (like triggers, transaction, etc.) and this will keep the PySBQL language simple.

6. Conclusions

The PySBQL language is deprived of “semantic reefs”. It is easy to use and results in a readable code. It is a language very accessible, especially for the Python programmers. Combining a query and a classical programming language,

PySBQL allows for efficient implementation of applications. It is dedicated for database applications. Thanks to the syntax similar to Python it has ability to import a number of Python's libraries and codes.

References

[1] Subieta K., Beeri C., Matthes F., Schmidt J.W., *A Stack-Based Approach to Query Languages*, (1993).
[2] Subieta K., *Theory and Construction of Object-Oriented Query Languages*. PJIIT – Publishing House, ISBN 83-89244-28-4, (2004), in Polish.
[3] www.python.org
[4] Kozankiewicz H., Leszczylowski J., Subieta K., *Updateable XML Views*. Proccedings of ADBIS'03, Springer LNCS 2798, (2003) 385.
[5] Kozankiewicz H., *Updateable Object Views*. PhD Thesis, 2005, http://www.ipipan.waw.pl/~subieta/ -> Finished PhD-s -> Hanna Kozankiewicz.