Universal Query Language

Piotr Wiśniewski and Krzysztof Stencel

Faculty of Mathematics and Computer Science
Nicolaus Copernicus University
Chopina 12/18, 87-100 Toruń
{pikonrad,stencel}@mat.umk.pl

Abstract. In the prequel paper we introduced the Unified State Model (USM), i.e. a single model that allowed conveying objects of popular programming languages and databases. That model exploited and emphasized common properties of all these objects. We showed mappings between those popular data models and USM. Our natural next research goal is the Unified Query Language (UQL). UQL is intended to be a minimalistic and elegant query language that allows expressing queries of major query languages. We plan to build a concise set of operators and show its coverage for mentioned languages' constructs. In this paper we present our initial efforts towards UQL. We present a set of language operators. We also provide a number of redolent examples of queries expressed in UQL.

1 Introduction

Our research originates from the so called object-relational impedance mismatch [1–3]. The relational model proposed in [4] eventually became dominant in data storage and nothing seems be a sign of a change in this respect. On the other hand already for a couple of decades applications are mainly written using object-oriented methods [5]. It soon turned out that object methods encounter nontrivial difficulties with the persistent relational data storage. Problems of impedance mismatch manifest themselves as (1) the mismatch of the data models, (2) the mismatch of the binding time, (3) the mismatch of the lifecycle. The impedance mismatch is not only related to the abovementioned paradigms. It also occurs when trying to interconnect object-oriented programming languages and other database systems like XML stores [6].

In our prequel paper [7] we focused on the mismatch of data models. We analysed data models of major programming languages and database systems. Using results of this analyses we proposed the Unified State Model (USM) that allows smooth mapping of the analysed data models. The conclusion of [7] is that although there are minor discrepancies, data models of modern object-oriented programming languages (e.g. Java and Python) and relational databases are noteworthy similar. Thus the object-relational mapping systems for such languages are feasible. The software market contains a number of proofs of this statement (e.g. Hibernate [8] for Java, LINQ [9] for .NET language family, django-models [10] for Python).

Having the data model of major programming and storage environments (USM), we were naturally tempted to pose a new research question. Is it possible to define a Universal Query Language (UQL) on the USM? UQL should subsume query languages for models that were subsumed by USM. This will supply evidence that also mapping queries between major programming and storage systems is a not miracle but a natural consequence of numerous similarities.

In this paper we show initial results of our research, i.e. a preliminary list of UQL operators and a number of representative examples of queries and their expression in UQL. Of course, the naturalness of the example mapping prove nothing. However, it suggests that building UQL is feasible. The next genuine step will be to establish a full set of UQL operators and to develop proofs that UQL subsumes query part of SQL, OQL [11], SQBL [12] and XQuery [13].

In fact, the success of Hibernate, observed on the market place, and object-relational mapping systems show that after C++ ceased to be the programming language of the first choice in favour of e.g. Java, Python and the C#, the development of object-relation mapping became significantly easier.

The paper is organized as follows. In Section 2 we sketch basic definitions of the Universal State Model (USM). Section 3 enumerates operators of the preliminary version of the Universal Query Language (UQL). In Section 4 we show a number of example queries and disccus their smooth mapping to UQL. Section 5 concludes.

2 Universal State Model

In this Section we remind some definition of the Unified State Model (USM) introduced in [7]. The Universal Query Language proposed in this paper processes objects of USM.

We use the following symbols. V is a set of atomic values. We assume that it contains all simple values. $N \subset V$ is a set of *external* names of objects with a chosen value $\eta \in N$ to tag nameless objects. External names are used by designers and programmers to access objects. Nameless objects can only be accessed through object references. $I \subset V$ is a set of *identities* with a chosen value **nil** $\in I$ to represent empty identifier/address. We assume that $N \cap I = \emptyset$.

For any set S we denote by $S^* = \bigoplus_{t \in \mathcal{N}} S$, i.e. the set of all finite sequence of elements of S.

Definition. *A state of an object of the level* 0 is a triple $(i, n, v) \in I \times N \times V$. By O_0 we mean the set of all states of the level 0. This set is also denoted by S_0.

Definition. Let $t \in \mathcal{N}$ be a positive natural number and assume that we have defined states of objects of the level k for all $k < t$ and S_{t-1} denotes the set of all states of objects of the levels lesser than t. A state of an object of the level t is a triple $(i, n, L) \notin S_{t-1}$ where $i \in I, n \in N$ and $L \in S_{t-1}^*$ is a finite sequence of elements of S_{t-1} and $L \notin S_{t-2}^*$. We assume that $S_{-1} = \emptyset$. The set of all states of objects of the level t will be denoted by O_t. By $S_t = O_t \cup S_{t-1}$ we denote the

set of all states of objects of the levels not greater than t. Members of the list L will be called states of subobjects of the object state o.

Each object has a name, an identifier and a value. This value can be simple (like in objects of the level 0) or be a list of objects. According to the above definition an object of level $t > 0$, i.e. a member of O_t, has at least one subobject of level $t-1$. Therefore, the level of an object is the maximum subobject nesting depth in this object.

Definition. S_∞ is the set of all states of objects of all natural levels.

In [7] we have formally defined the notion of *proper* states. In a proper state no object identity is repeated and all object references are valid, i.e. there are no dangling pointer objects.

Since in Section 4 we use SQL queries to illustrate concepts of UQL, below we also remind some information on the mapping between USM and the relational model presented in [7]. Our analysis of the relational data model is based on the multi-set extended relational algebra [14].

Let $\mathcal{R}$ be a relational schema, also called the type of a relation denoted by $dom(R)$ such that $dom(R) = dom(A_1) \times ... \times dom(A_n)$, where $dom(A) \subset V$ is some domain.

We define a tuple of a relation of type $dom(R)$ as a state of an object $o = (i, R, o_1, ..., o_n)$ such that for $k = 1, ..., n$ $o_k = (i_k, A_k, v_k)$ is an atomic object such that $v_k \in dom(A_k)$. In particular, a tuple $o \in O_1$ is a state of the first level. Let us recall a definition from [7]:

Definition. The tuples $o = (i, R, o_1, ..., o_n), o' = (i', R, o'_1, ..., o'_n)$ shall be called *relationally equal* if for all $k = 1, ..., n$ it holds that $v_k = v'_k$.

Let us consider the table `emp` with five columns `empno`, `firstname`, `lastname`, `salary`, and `deptno`. The two object states o and o' represent the tuple (1, "Jan", "Kowalski", 2) of this table. Although they are different, they are relationally equal:

$$
\begin{aligned}
o \quad = (i, emp, \{ & \\
& (i_0, empno, 1) \\
& (i_1, firstname, \text{"Jan"}) \\
& (i_2, lastname, \text{"Kowalski"}) \\
& (i_3, deptno, 2) \\
& (i_4, salary, 3000) \\
\}) \in S_1 &
\end{aligned}
$$

$$
\begin{aligned}
o' = (i', person, \{ & \\
& (i'_0, empno, 1) \\
& (i'_1, firstname, \text{"Jan"}) \\
& (i'_2, lastname, \text{"Kowalski"}) \\
& (i'_3, deptno, 2) \\
& (i'_4, salary, 3000) \\
\}) \in S_1 &
\end{aligned}
$$

Next we deal with relations that are defined by Grefen [14] as multisets of tuples. The state of a relation R is any object (i, R, T), where T is a sequence of tuples of the type R.

Definition. Let the states $o = (i, R, T) \in O_2$ and $o' = (i', R, T') \in O_2$ be of the second level, where T and T' are sequences of states of the first level. Assume also that $o \approx_{\mathcal{R}} o'$. States o and o' are called relationally equal iff $|T| = |T'|$ and there is a permutation p of the sequence T, such that for each $t = 1, ..., |T|$ the states of the tuples $p(T)_t$ and T'_t are relationally equal.

Note that states of the second level from the above definition model multisets of tuples and the *relational equality* of states defined above is an equivalence relation.

3 UQL Operators

In this Section we introduce operators of the Universal Query Language (UQL). Let $\mathbf{S^S}$ be the set of all functions $Q : \mathbf{S} \to \mathbf{S}$. Queries of UQL are functions from the set $\mathbf{S^S}$. These functions are compositions of UQL operators defined below.

3.1 Renaming

The operator as gives an object a new name:

$$as : N \to \mathbf{S^S}$$
$$as(n)((i, n', v)) = (i, n, v)$$

3.2 Flattening

For a complex object the operator $flat$ replaces its subjects with the contents of these subobjects. If the argument complex object is simple or has any simple subjects, $flat$ returns an empty object. Assume $o = (i, n, (o_1, o_2, \ldots, o_k))$. If for all $t = 1 \ldots k$ $o_t \notin O_0$ and $o_t = (i_t, n_t, (o_{t,1}, \ldots, o_{t,j_t})$, then

$$flat(o) = (i, n, (o_{1,1}, o_{1,2}, \ldots, o_{1,j_1}, o_{2,1}, \ldots, o_{k,j_k}))$$

Otherwise:

$$flat(o) = (i, n, nil)$$

3.3 Mapping

The purpose of the operator map is the application of a function to all subobjects of the given object. The functor $map : \mathbf{S^S} \to \mathbf{S^S}$ is defined as follows. Let $f : \mathbf{S} \to \mathbf{S}$ be the function to be applied. Then:

$$map(f)(o) = \begin{cases} (i, n, (f(o_1), \ldots, f(o_n)) & \text{if } o = (i, n, (o_1, o_2, \ldots, o_k)) \notin O_0 \\ (i, n, nil) & \text{if } o \in O_0. \end{cases}$$

3.4 Evaluating

Given a function on simple values we can apply it to the simple values stored inside an object. We introduce a family of functors *eval*. For a given natural t $eval_t$ maps any function $f : V^t \to V$ to the query $eval_t(f)$ such that:

$$eval_t(f)(i, n, (o_1, \ldots, o_k)) = \begin{cases} (i', \eta, f(v_1, \ldots, v_t)) & \text{if } k = t \text{ and} \\ & \quad o_j = (i_j, n_j, v_j) \in O_0 \\ (i', \eta, \mathbf{nil}) & \text{otherwise.} \end{cases}$$

3.5 Getting k-th subobject

Let $o = (i, n, (o_1, o_2, \ldots, o_t))$ be a state of an object. We define:

$$take_k(i, n, (o_1, \ldots, o_t)) = \begin{cases} o_k & \text{if } k < t \text{ and} \\ (i', \eta, \mathbf{nil}) & \text{otherwise.} \end{cases}$$

For $k = 1$ we will use notation $takeFirst$ instead of $take_1$.

3.6 Filtering

Filtering is also performed by a functor. Assume a function $p : \mathbf{S} \to V$ that returns Boolean values. Then:

$$filter(p)(o = (i, n, (o_1, o_2, \ldots, o_k))) = (i, n, (o_{i_1}, o_{i_2}, \ldots, o_{i'_k}))$$

where the sequence $(o_{i_1}, o_{i_2}, \ldots, o_{i'_k})$ is the subsequence of $(o_1, o_2, \ldots, o_k)$ composed of all objects o_j such that $eval_1(p(o_j)) = true$.

3.7 Nesting

The operator *nest* creates a new nameless object and puts the argument object into it:

$$nest(o) = (i', \eta, o)$$

3.8 Cloning

The operator $clone_t$ for natural $t > 1$ duplicates the state of the given object t times. This operator does not preserve the propriety of the object since the clones will have repeated identities inside. The operator $clone_t : \mathbf{S^S} \to \mathbf{S^S}$ is defined as follows:

$$clone_t(o = (i, n, x)) = (i', n, ((i_1, n, x_1), \ldots, (i_n, n, x_t)))$$

where $x_1, x_2, \ldots, x_t$ are fresh object identifiers.

3.9 Product

Let $o = (i, n, (o_1, ..., o_t))$, $o' = (i', n', (o'_1, ..., o'_{t'}))$ be complex objects. The *cross* product operator is defined as follows:

$$cross(o, o') = (i, \eta, ((o_{11}, ..., o_{1t'}, ..., o_{tt'}), \text{ where } o_{jk} = (i_{jk}, \eta, (o_j, o'_k))$$

3.10 Grouping

Let $\approx$ be an equivalence relation on a subset of states $S \subset S_\infty$. Let $o = (i, n, (o_1, ..., o_t))$ be a complex object such that $\{o_1, ..., o_t\} \subset S$. The operator *abstract* is defined as follows:

$$abstract_\approx(o) = (i', n, (u_1, ..., u_k))$$

where $u_j = (i_j, n, (o_{j1}, ..., o_{jt_j}))$ and the following conditions are satisfied:

a. The sequence $(o_{j1}, ..., o_{jt_j})$ is a subsequence of $(o_1, ..., o_t)$ for all $j = 1, \ldots, k$ and the sequence $o_{11}, ..., o_{k1}$ is a subsequence of $(o_1, ..., o_t)$.
b. The items in $(o_{j1}, ..., o_{jt_j})$ are pairwise related with respect to $\approx$.
c. For any $j \neq j'$ and m, m' it is true that $o_{jm} \not\approx o_{j'm'}$.

3.11 Transposing

Let $o = (i, n, (o_1, ..., o_m))$ be a complex state of an object, such that there exists a number t and a name n', such that for all $j = 1, ..., m$, we have $o_j = (i_j, n', (o_{j,1}, ..., o_{j,t}))$. In other words the state o represents a two dimensional $m \times t$ matrix of states of objects. We define the *transpose* operator as follows:

$$transpose(o) = (i', n, (o_1, ..., o_t)),$$

where $o_k = (i'_k, n', (o_{1,k}, ..., o_{t,k}))$, for $k = 1...t$

3.12 Folding

Let $o = (i, n, (o_1, ..., o_t))$ be an complex model and $f : S \times S \to S$ be a binary operator. We define $fold$ via induction:

$$fold_f(o = (i, n, \{o_1, ..., o_t\})) = \begin{cases} o & \text{, if } t = 1 \\ fold_f((i', n, (f(o_1, o_2), o_3, ..., o_t))) & \text{, if } t > 1 \end{cases}$$

4 Mapping example queries

In this Section we show some examples of SQL queries mapped to UQL. As we mentioned in the introduction, we do not treat them as a proof of the quality of UQL. However, the smoothness of the mappings in an indicator of the feasibility of our research goal.

In order to simplify the presentation we are going to use the following notation. When two unary operators op_1, op_2 are applied to a pair of objects, it is expressed as $(op_1 \otimes op_2)(o_1, o_2)$.

Let us assume that we have the following relational database schema with two well-known tables:

```
emp: empno, firstname, lastname, salary, deptno
dept: deptno, deptname, location
```

Consider the following SQL query:

```
SELECT firstname, lastname
FROM emp
WHERE lastname = 'Schmidt'
```

Let us assume that the variable o represents the state of the database with the above schema. Our example query gets the following form in UQL:

$$
\begin{array}{l}
map(\ filter_{name='firstname' \vee name='lastname'}(\\
\quad filter_{\exists on: name(on)='lastname' \wedge value(on)='Schmidt'}(\\
\quad\ flatten(\\
\quad\quad filter_{name='emp'}(o) \\
\quad\quad) \\
\quad\ ) \\
\quad) \\
)
\end{array}
$$

Now let us consider a join. The following query is a natural join with an equality selection:

```
SELECT first, lastname
FROM dept join emp using deptno
WHERE dept.deptname = 'IT'
```

This query is equivalent to:

```
SELECT e.first, e.lastname
FROM dept d, emp e
WHERE d.deptname = 'IT'
   AND d.deptno = e.deptno
```

Let us build the mapping step by step. Again we assume that the variable o is the state of the database. A *from* clause for a single table is realised as follows (we filter out other tables):

$$filter_{name='tablename'}(o)$$

Thus, the clause `from dept d` is mapped as (the operator as augments the column name with $d\_$):

$$map(as_{d\_+name}, filter_{name='dept'}(o))$$

Furthermore, `from dept d, emp e` is expressed as:

$$\begin{array}{l}(map(as_{d\_+name}, \ldots) \otimes map(as_{e\_+name}, \ldots))(\\ \quad (filter_{name='dept'} \otimes filter'_{name='emp'})(\\ \quad\quad duplicate_2(o) \\ \quad\quad) \\ \quad)\end{array}$$

Now the obtained objects are glued together:

$$flatten(cross(\ldots))$$

and filtered:

$$filter_{val(d\_deptno)=val(e\_deptno) \wedge val(d\_deptname)='IT'}(\ldots)$$

Finally, we have to select the columns that constitute the result of the query:

$$map(filter_{name='firstname' \vee name='lastname'},)$$

Altogether, our query formulated in UQL has the following form:

$$\begin{array}{l}map(filter_{name='firstname' \vee name='lastname'}, \\ \quad flatten(cross(\\ \quad\quad (map(as_{d\_+name}, .) \otimes map(as_{e\_+name}, .))(\\ \quad\quad\quad (filter_{name='dept'} \otimes filter'_{name='emp'})(\\ \quad\quad\quad\quad duplicate_2(o) \\ \quad\quad\quad\quad) \\ \quad\quad\quad) \\ \quad\quad)) \\ \quad)\end{array}$$

Note that this construction of standard SPJ queries is quite regular. We start from the *from* clause, then goes *where* and finally *select*. Note that in all three steps most of the effort is in fact filtration. The process is compositional, so the mapping of subqueries will be similarly smooth. This way we can map other query languages:

```
SBQL: emp where lastname = 'Schmidt'
HQL:  from emp where lastname = 'Schmidt'
SQL:  SELECT * from emp where lastname = 'Schmidt'
```

All of these queries simply map to the following UQL query:

$$filter_{value(lastname)='Schmidt'}(flatten(filter_{name='emp'}))$$

Finally, let us consider the aggregation. The example query follows:

```
SELECT deptno, SUM(salary)
FROM emp
GROUP BY deptno
```

Like previously, at the beginning we need to select the *emp* objects, flatten them and filter out uninteresting subobjects:

$$\begin{aligned} selo = map(&filter_{name='deptno' \vee name='salary'}, \\ &flatten(filter_{name='emp'}(o)) \\ ) \end{aligned}$$

In the next step we cluster resulting objects using relation *dn*. Two objects are related by *dn*, if their subobjects *deptno* have the same value:

$$grpo = abstract_{dn}(selo)$$

The subobjects of *grpo* contain collections of pairs, but in order to $fold$ them we need pairs of collections, so we transpose the matrices:

$$trso = map(transpose, grpo)$$

Then we $fold$ flattened states:

$$map(takeFirst \otimes (flaten \circ fold_{+}), trso)$$

Eventually we have received a state containing a collection of pairs composed of *deptno* and the sum of *salary*. Thus, the example aggregation query maps to:

$$\begin{aligned} &map(takeFirst(\otimes flaten \circ fold_{+}), \\ &\ map(translate, \\ &\ \ abstract_{dn}(\\ &\ \ \ map(filter_{name='deptno' \vee name='salary'}, \\ &\ \ \ \ flatten(filter_{name='emp'}(o)) \\ &\ \ \ ) \\ &\ \ ) \\ &\ ) \\ ) \end{aligned}$$

The construction of this UQL query has been stepwise and natural. We are convinced that even complex SQL (and other QL as well) queries will smoothly map to UQL.

5 Conclusions and future work

After successfully constructing USM (the Unified State Model), i.e. the unified model for a number of major programming and storage environments, we were naturally excited, if it is possible to build a Universal Query Language (UQL) on the USM? UQL is planned to subsume query languages for models that were covered by USM.

In this paper we made an initial attempt to answer this question. We presented a pilot list of UQL operators and a number of representative examples of queries and their expression in UQL. We chose two SPJ queries and one aggregate query, showed their smooth mapping to UQL and concluded that such queries seem to map well. Of course, this proves nothing. However, it indicates that building UQL is feasible.

The next step planned in our research it to found a final robust set of UQL operator and show that UQL allows expressing the query part of SQL, OQL, SQBL, XQuery and other interesting query languages.

References

1. Neward, T.: The Vietnam of computer science. Online (2006)
2. Neward, T.: Avoiding the quagmire. Online (2007)
3. Hughes, S.: Object relational mapping; how Vietnam can still be won. Online (2008)
4. Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13** (1970) 377–387
5. Coad, P., Nicola, J.: Object-Oriented Programming. Yourdon Press (1993)
6. Laemmel, R., Meijer, E.: Revealing the X/O impedance mismatch. In Backhouse, R., Gibbons, J., Hinze, R., Jeuring, J., eds.: Datatype-Generic Programming. Volume 4719 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2007) 285–367
7. Wiśniewski, P., Burzańska, M., Stencel, K.: The impedance mismatch in light of the unified state model. Fundam. Inform. (2012, to appear)
8. O'Neil, E.J.: Object/relational mapping 2008: Hibernate and the Entity Data Model (EDM). In Wang, J.T.L., ed.: SIGMOD Conference, ACM (2008) 1351–1356
9. Meijer, E.: The world according to LINQ. Queue **9** (2011) 60:60–60:72
10. Forcier, J., Bissex, P., Chun, W.: Python Web Development with Django. 1 edn. Addison-Wesley Professional (2008)
11. Cattell, R.G.G., Barry, D.K.: The Object Data Standard: ODMG 3.0. Morgan Kaufmann (2000)
12. Card, M.: OMG next-generation object database standardization white paper. Online (2007)
13. W3C: XQuery 1.0: An XML Query Language. (2005)
14. Grefen, P.W.P.J., de By, R.A.: A multi-set extended relational algebra - a formal approach to a practical issue. In: ICDE, IEEE Computer Society (1994) 80–88