Object-Oriented Programming with Recursive Queries

Tomasz Pieciukiewicz[1], Krzysztof Stencel[2], Kazimierz Subieta[1,3]

[1] Polish-Japanese Institute of Information Technology, Warsaw, Poland
{pietia, subieta}@pjwstk.edu.pl
[2] Institute of Informatics, Warsaw University, Warsaw, Poland
stencel@mimuw.edu.pl
[3]Institute of Computer Science PAS, Warsaw, Poland

Abstract. Recursive queries are required in many object-oriented database applications. Among them we can mention Bill-Of-Material (BOM), various kinds of networks (transportation, telecommunication, etc.), processing semi-structured data (XML, RDF), and so on. The support for recursive queries in current query languages is limited and lacks theoretical foundations. In this paper we present recursive query processing capabilities for object-oriented environments. They are part of Stack-Based Query Language (SBQL). SBQL offers powerful and flexible recursive querying capabilities due to the fact that recursive processing operators are fully orthogonal to other features of this language. This paper discusses corresponding SBQL constructs: variants of transitive closures, fixed point equations and recursive procedures/views. The paper is augmented by discussion concerning the state-of-the-art of current recursive querying capabilities.

1 Introduction

Recursion in traditional programming languages like C or Java is natural and obvious for all the programmers. Database programmers and users, however, usually do not have the possibility of utilizing recursion, despite frequent needs to use it in database applications. The most widely known task requiring recursive processing is Bill-Of-Material (BOM), which is a part of Materials Requirements Planning (MRP) systems. BOM acts on a recursive data structure representing a hierarchy of parts and subparts of some complex material products. Typical MRP software processes such structures by proprietary routines and applications implemented in a traditional programming language. However, users frequently need to issue *ad hoc* queries addressing such structures. In such cases they need special recursive user-friendly facilities of a query language. Similar problems concern computations on genealogic trees, stock market dependencies, various types of networks (transportation, telecommunication, electricity, gas, water, and so on), processing metadata such as CORBA Interface Repository, etc.

In traditional programming languages in many cases recursion can be substituted by iteration. This, however, implies much lower abstraction level and less elegant problem specification. The iteration may also cause higher cost of program maintenance,

since it implies a clumsy code, more difficult to debug and change. Sometimes iteration instead of recursion is motivated by attempts to achieve higher performance, although nowadays, due to very high speed of computers, the performance gain is very rarely worth extra maintenance cost. Another motivation for changing recursion into iteration is the limit on the size of the stack, but for current sizes of memories (extended by virtual memories) this limit is not critical for majority of recursive tasks. Database programmers cannot use iteration in a query language, since it is available only within a host programming language embedding or integrating the query language.

Despite importance, recursion is not supported in SQL standards (SQL-89 and SQL-92). Beyond the standards, it is implemented (differently) in relational database management systems, in particular, in Oracle and DB2, in the form of transitive closures. Newer SQL standards SQL-99 (aka SQL-3) and SQL 2003 introduce both transitive closure and deductive rules *a la* Datalog. Unfortunately these standards are very huge and eclectic, thus many database professionals doubt if they will ever be fully implemented. The ODMG standard for object-oriented databases and its query language OQL do not mention any corresponding facilities. The recursion is considered a desirable feature of XML-oriented and RDF-oriented query languages, but current proposals and implementations do not introduce corresponding features or introduce them with many limitations.

The possibility of recursive processing has been highlighted in the field of deductive databases, notably Datalog. The paradigm has roots in logic programming and has several variants. Some time ago it was advocated as a true successor of relational databases, as an opposition to the emerging wave of object-oriented databases. Despite high hype and pressure of academic communities it seems that Datalog falls short of the software engineering perspective. It has several recognized disadvantages, in particular: flat structure of programs, limited data structures to be processed, no powerful programming abstraction capabilities, impedance mismatch during conceptual modeling of applications, poor integration with typical software environment (e.g. class/procedure libraries) and poor performance. Thus practical mission-critical Datalog applications are till now unknown. Nevertheless, the idea of Datalog semantics based on fixpoint equations seems to be very attractive to formulate complex recursive tasks. Note however that fixpoint equations can be added not only to languages based on logic programming, but also to any query language, including SQL, OQL and XQuery.

Besides transitive closures and fixpoint equations there are classical facilities for recursive processing known from programming languages, namely recursive functions (procedures, methods). In the database domain a similar concept is known as recursive views. Integration of recursive functions or recursive views with a query language requires generalizations beyond the solutions known from typical programming languages or databases. First, functions have to be prepared to return bulk types which a corresponding query language deals with, i.e. a function output should be compatible with the output of queries. Second, both functions and views should have parameters, which could be bulk types compatible with outputs of queries. Currently very few existing query languages have such possibilities, thus using recursive functions or views in a query language is practically unexplored.

This paper discusses three different approaches to recursive querying:

- transitive closure operators,
- least fixed point equation systems (fixpoint equations, for short),
- recursive procedures and views.

In this paper we describe all three approaches to recursive processing within a unified framework: the Stack Based Approach (SBA) to object-oriented query languages [1, 2, 3]. SBA treats a query language as a kind of programming languages and therefore, queries are evaluated using mechanisms which are common in programming languages. SBA introduces an own query language Stack-Based Query Language (SBQL) based on abstract, compositional syntax and formal operational semantics. Within this approach all three approaches can be implemented. Currently we have implemented several variants of transitive closure operators and recursive procedures/views. Implementation of fixpoint equations is advanced in the currently developed object-oriented database platform ODRA. In this paper we compare the approaches on sufficiently complex examples showing their strengths and weakness with respect to problems from database application programming.

The rest of the paper is organized as follows. Section 2 presents the Stack-Based Approach and its query language SBQL. Section 3 is devoted to the transitive closure in SBQL. Section 4 describes fixpoint equations in SBQL. Section 5 deals with recursive procedures and views. Section 6 discusses the future work on optimization. Section 7 presents a short overview of the current state of the art in recursive queries. Section 8 concludes.

2 Stack Based Approach and Stack Based Query Language

The Stack-Based Approach (SBA) [1, 2, 3] is a consistent approach to theory and construction of query languages for various data models. It is an attempt to discipline the current creative chaos in the domain on the ground of a homogeneous and universal theory. In SBA a query language is considered a kind of a programming language. Thus, the semantics of queries is based on concepts well known from programming languages like the environmental stack (thus the name of the approach) and the naming-scoping-binding principle. SBA extends the stack mechanism for the case of query operators, such as selection, projection/navigation, join, quantifiers, transitive closures and others. Using SBA one is able to determine precisely the operational semantics (abstract implementation) of query languages, including relationships with object-oriented concepts, embedding queries into imperative constructs, and embedding queries into programming abstractions: procedures, functional procedures, views, methods, modules, etc.

The stack-based approach is a theory of query languages that is independent of a specific data model. It can be applied to relational, object-oriented, object-relational databases, and to XML repositories. SBA introduces the query language SBQL (Stack-Based Query Language), which has abstract syntax and rigorous, clean formal semantics. SBQL is incomparably more powerful than ODMG OQL and W3C XML

query languages such as XQuery. SBQL, together with imperative extensions and abstractions, has the computational power of programming languages, similarly to Oracle PL/SQL or SQL-99/SQL-2003 standards. Due to object relativism, full orthogonality and compositionality, and other qualities, the specification of SBQL has some 100 pages, i.e. it is more than 20 times shorter than the specification of SQL-99.

SBA and SBQL can also be considered a theoretical frame in the spirit of the relational algebra and relational calculus known from the relational model and their object-oriented counterparts, such as object algebras and F-logic. In comparison, however, SBQL offers much wider conceptual basis concerning both data structures to be queried and query operators. Clean, rigorous semantics of SBQL, supporting the principle of object relativism and full orthogonality of SBQL operators create a big potential for query optimization and for strong static type checking of SBQL queries and programs integrated with SBQL queries.

SBA and SBQL have already several implementations, in particular, within the prototype of object-oriented DBMS LOQIS, for the European project ICONS (Intelligent CONtent management System), as a query language supporting the workflow definition language XPDL, for the object-oriented DBMS Objectivity/DB, and recently for a new prototype ODRA (Object Database for Rapid Application development) aiming at Web content management and grid technologies developed for the .NET environment.

2.1 Data Store Models

Any approach to formalization of query languages must be preceded by formalization of data structures to be queried. Object-oriented data models are complex and introduce a lot of notions. The question how to formalize them is not trivial, because the formalization must be simple formally and simultaneously must cover (at least potentially) all critical concepts of object databases. A trade-off is presented in [1, 2, 3] and other papers devoted to SBA. To eliminate secondary features of data structures we assume a unification of records, tuples, arrays and all bulk structures. In our models we abstract from their differences. Similarly, we unify many other concepts related to database models and object-orientedness without sacrificing their conceptual qualities.

In the Stack Based Approach four data store models are defined, with increasing functionality and complexity. The M0 model described in [2] is the simplest data store model. In M0 objects can be nested (with no limitations on nesting levels) and can be connected with other objects by links. M0 covers relational and XML-oriented structures. It can be easily extended to comply with more complex models which include classes and static inheritance (M1), dynamic object roles and dynamic inheritance (M2), encapsulation (M3) and other features of object-oriented databases.

In SBA an object has the following properties: *internal object identifier* (OID) which cannot be used in queries nor printed, *external name*, which is used in the application code to access objects, and object content which can be a value, a link, or a set of objects. An SBA store consists of the structure of such objects/subobjects and the set of identifiers of root objects, i.e., starting points for queries.

In the M1 model a *class* is a special kind of an object stored in the database. It is a complex object which contains invariants of all objects of this class. These invariants include codes of methods, default values of attributes, the common name for all objects of the class. Two important relations are stored in the database. The first one is the memberships relation which connects objects to classes. The second one is the inheritance relation which connects classes to classes.

2.2 Name Binding and Environment Stack

SBA is based on the programming languages' naming-scoping-binding principle. Each name occurring in a query/program is bound to a proper run-time database/program entity according to the name scope. Scopes for names are managed by means of the Environment Stack (ES). ES consists of sections which contain entities called binders. Binders relate names with run-time objects and are used during binding names. A *binder* is a pair (n, v), written as $n(v)$, where n is an external name used in queries and v is a value (most often it is an object identifier).

New sections on ES are built by means of a special function *nested* which returns binders to the content of an object (in case of complex objects) or (in case of link objects) a binder to the pointed object.

Binding name n occurring in a query is an action of the query interpreter which searches ES for the binder named n which is closest to the top of ES. Binding respects static scoping rules which mean that some sections of ES are invisible during the binding (e.g. sections related to local environments of procedures). The name binding can return multiple binders and this way we handle collections of objects.

The M1 model requires a slight modification of the binding mechanism so that the *substitutability* principle is obeyed. Whenever the being bound name m is the name of objects of a class c, the binding returns all the objects named m but also all the objects of the subclasses of the class c. Full description of the binding mechanism in M1 can be found in [3].

2.3 Stack-Based Query Language (SBQL)

Stack-Based Query Language [2, 3] is based on the principle of compositionality, i.e. semantics of a complex query is recursively built from semantics of its components. SBQL queries are defined as follows:

1. A name or a literal is a query; e.g., 2, "Niklaus Wirth", *Book*, *author*.
2. σq, where σ is a unary operator and q is a query, is a query; e.g., *count*(*Book*), *cos*(x).
3. $q_1 \ \tau \ q_2$, where τ is a binary operator, is a query; e.g., 2+2, *Book.title*, *Customer* **where** (*name* = "Smith").

In SBQL each binary operator is either algebraic or non-algebraic. If Δ is an algebraic operator, then in the query $q_1 \ \Delta \ q_2$ the order of evaluation of queries q_1 and q_2 is inessential. Queries are evaluated independently and their results are combined into

the final result depending on Δ. Examples of algebraic operators are numerical and string operators and comparisons, aggregate functions, union, and others.

Non-algebraic operators are the core of the SBA. In a query q_1 θ q_2 with a non-algebraic operator θ the second subquery is evaluated in context determined by the first subquery. Thus the order of evaluation of queries q_1 and q_2 is significant. Query q_1 θ q_2 is evaluated as follows. First q_1 is evaluated. Then q_2 is evaluated for each element r of the result returned by q_1. Before each such evaluation ES is augmented with a new scope determined by *nested*(r). After evaluation the stack is popped to the previous state. A partial result of the evaluation is a combination of r and the result returned by q_2 for this value. The method of the combination depends on θ. Eventually, these partial results are merged into the final result depending on the semantics of operator θ. Examples of non-algebraic operators are selection (**where**), projection/navigation (the dot), dependent join, quantifiers (∃, ∀), sorting (order by), and transitive closures.

3 Transitive Closures in SBQL

A transitive closure in SBQL is a non-algebraic operator having the following syntax**:**

q_1 **close by** q_2

Both q_1 and q_2 are queries. The query is evaluated as follows. Let *final_result* be the final result of the query and ∪ the bag union. Below we present the pseudo-code accomplishing abstract implementation of q_1 **close by** q_2:

final_result := *result_of* (q_1);
for each $r \in$ *final_result* do:
    push *nested*(r) at top of ENVS.
    final_result := *final_result* ∪ *result_of* (q_2);
    pop ENVS;

Note that each element r added to *final_result* by q_2 is subsequently processed by the *for each* command. The above operational semantic can be described in the denotational setting as the least fixed point equation (started from *final_result* = ∅ and continued till fixpoint):

$$final\_result = q_1 \cup final\_result.q_2$$

where dot is identical with the dot operator in SBQL. Similarly, the semantics can be expressed by iteration (continued till *result_of* (q_2) = ∅):

$$final\_result = q_1 \cup q_1.q_2 \cup q_1.q_2.q_2 \cup q_1.q_2.q_2.q_2 \cup ....$$

Naive implementation of the **close by** operator is as easy as the implementation of the dot operator. Note that if q_2 returns a previously processed element, an infinite loop will occur. Checking for such situations in queries is sometimes troublesome and

may introduce unnecessary complexity into the queries. Another operator **distinct close by** has been introduced to avoid infinite loops due to duplicates returned by q_2.

As q_1 and q_2 can be any queries, simple or complex, the relation between elements which is used for transitive closure is calculated on the fly during the query evaluation; thus the relation needs not to be explicitly stored in the database.

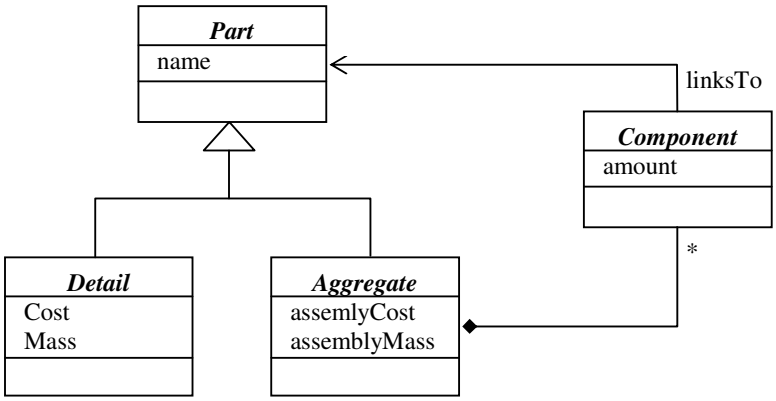

Fig. 1. A sample data schema

Fig.1 depicts a simple data schema used in our examples. It is a description of parts, similar to descriptions used in Bill of Material (BOM) applications. Each *Part* has a *name*. The part may be either a *Detail* or *Aggregate*. Each detail has attributes *Cost* and *Mass* (its cost and mass). Each *Aggregate* has attributes *assemblyCost* and *assemblyMass* which represent the cost of assembling this aggregate and mass added to the mass of its components as the result of the assembly process. Aggregates have one or more *Component* subobjects. Each *Component* has the attribute *amount* (number of components of specific type in a part), and a pointer object *leadsTo* with one-direction navigation to *Parts*.

The following SBQL query with a transitive closure over this schema finds all components of a part named “engine”.

(*Part* **where** *name* = ”engine”) **close by** (*Component.leadsTo.Part*)

This query first selects parts having the attribute *name* equal to “engine”. The transitive closure relation is described by the subquery (*Component.leadsTo.Part*). It returns all *Part* objects which can be reached by the pointer *leadsTo* from already selected objects.

This query takes advantage of an assumption made in the M1 model. After reaching an object by its name all its attributes are accessible, not only those of the class whose name has been used to retrieve the object. The assumption allows us to simplify queries by removing the necessity to use cast operators. If an object does not have the queried attribute (because it belongs to a subclass which does not have this attribute or the attribute is optional), the empty collection is returned as the result of binding the name of this attribute. In this query, the subquery (*Component.leadsTo.Part*) is evalu-

ated for *Part* objects which can be either *Details* or *Aggregates*. In case of an *Aggregate*, the name *Component* is properly bound and returns the collection of the *Component* subobjects. In case of *Detail*, the name *Component* cannot be bound and returns the empty set.

One of the basic BOM problems, i.e. "find all components of a specific part, along with their amount required to make this part", may be formulated using the transitive closure as follows:

((*Part* **where** *name*="engine"), (1 **as** *howMany*))
close by (*Component*.((*leadsTo.Part*), (*howMany***amount*) **as** *howMany*))

The query uses a named value in order to calculate the number of components. The number of parts the user wants to assemble (in this case 1) is named *howMany* and paired with the found part. In subsequent iterations the *howMany* value from parent object is used to calculate the required amount of child elements. It is also named *howMany* and paired with the child object.

The above query does not sum up amounts of identical sub-parts from different branches of the BOM tree. Below we present a modified query which returns aggregated data – sums of distinct components from all branches of the BOM tree:

```
( (    ((Part where name="engine") as x, (1 as howMany))
       close by (Component.((leadsTo.Part) as x, (howMany*amount) as howMany))
  ) group as allEngineParts
).
((distinct(allEngineParts.x) as y).(y, sum((allEngineParts where x=y).howMany)))
```

This query uses grouping in order to divide the problem into two parts. First, all the components named *x*, along with their amounts named *howMany* are found. The pairs are then grouped and named *allEngineParts*. The grouped pairs are further processed, by finding all distinct elements and summing the amounts for each distinct element.

This query could be further refined, in order to remove all aggregate parts (so only the detail parts will be returned). There are many ways to accomplish this goal. On of them is to use the operator **leaves by** in place of **close by**. The operator **leaves by** returns only leaf objects, i.e. objects which do not result in adding any further objects to the result set:

```
( (    ((Part where name="engine") as x, (1 as howMany))
       leaves by(Component.((leadsTo.Part) as x, (howMany*amount) as howMany))
  ) group as allEngineDet
).
((distinct(allEngineDet.x) as y).(y, sum((allEngineDet where x=y).howMany)))
```

The other way to sort the aggregates out of the result of the previous query is to use a cast. The cast operator takes a collection of objects and returns only these items of it which belong to the given class. The cast applied to a single object returns this object if it belongs to the class; otherwise it returns the empty set. Here is a query with the same retrieval goal but written using a cast.

```
( ( (Detail)
    (
     ((Part where name="engine") as x, (1 as howMany))
     close by (Component.((leadsTo.Part) as x, (howMany*amount) as howMany))
    )
  ) group as allEngineDet
).
((distinct(allEngineDet.x) as y).(y, sum((allEngineDet where x=y).howMany)))
```

Here the result of the subquery whose result is further named *allEngineDet* is first cast to the class *Detail*. This cast drops all objects which do not belong to this class.

Such rather typical BOM tasks cannot be formulated in any variant of SQL as a single query. Although the complexity of the SBQL solution is still high, SBQL supports facilities to manage the complexity. In this case the grouping operator allows us to decompose the problem into easier subproblems.

SBQL queries may be used to perform even more complex tasks. The query below calculates the cost and mass of the part named "engine", taking into account cost and mass of each engine part, amount of engine parts and cost and mass increment connected with assembly. This task has been used in [4] as an example of lack of power and flexibility of currently used query languages. In SBQL the task can be formulated with no essential problems:

```
(   (   ((Part where name="engine") as x, (1 as howMany))
        close by Component.((leadsTo.Part) as x, (amount*howMany) as howMany)
    ) group as allEngineParts
).
(   allEngineParts.((Detail) x as d).
                              ((howMany * d.Cost) as c, (howMany * d.Mass) as m)
                                                    group as CostMassIncreaseD,
    allEngineParts.((Aggregate) x as a).
                ((howMany * a. assemblyCost) as c, (howMany * a. assemblyMass) as m)
                                                    group as CostMassIncreaseA
).
(   (sum(CostMassIncreaseD.c)+ sum(CostMassIncreaseA.c)) as engineCost,
    (sum(CostMassIncreaseD.m)+ sum(CostMassIncreaseA.m)) as engineMass)
```

Due to the orthogonality (including orthogonal persistence) SBQL can perform calculations without referring to the database; e.g. 2+2 is a regular query. It is impossible in some SQL variants. As an example of the SBQL power, the query below calculates approximation of the square root of a, using the fixpoint equation $x = (a/x + x)/2$.

```
( (1 as x, 1 as counter)
  close by (((a/x + x)/2 as x, counter +1 as counter) where counter ≤ 5)
).(x where counter = 5)
```

Cycles in the queried graph can be easily dealt with by means of another variant of the **close by** operator – **close unique by**. This variant removes duplicates after each closure iteration, thus cycles do not imply infinite loops. Another variant of the **close**

by operator is the **leaves unique by** operator. It is a combination of the two previous variants. It returns only leaf objects, while preventing problems with cycles in graphs.

We note that cycles in the queried graph do not have to be an effect of database inconsistency. It is easy to imagine a database, which contains a graph with cycles that is consistent with the real world situation, Fig.2.

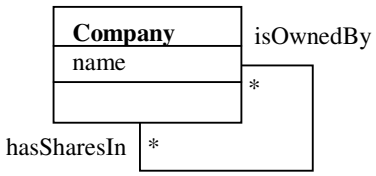

Fig. 2. A sample structure for the examples of an application of **close unique by**

A *Company* has *name* and may own shares in other *Companies* (in this example we abstract from the amount of shares). Other companies in turn may own shares in further companies; and so on. It is possible for a company (lets call it "ACME") to own shares in another company which in turn owns shares in "ACME" (directly or by owning shares in other companies). This creates cycles in the graph. To find the names of all companies owned (directly or indirectly) by "ACME" we cannot use a simple query using the **close by** operator as it would not function correctly upon encountering the cycle. We can use the **close unique by** operator instead, as shown in the example below:

(*Company* **where** *name* = "ACME") **close unique by** (*hasSharesIn.Company*)

The SBQL transitive closure operator is orthogonal to other operators of the language, it is very universal and well suited for querying. However, as shown above, some advanced tasks may lead to very complex queries which semantics could be difficult to grasp for the programmers. Such complex problems may be solved by means of other recursive querying techniques which support easier problem decomposition.

4 Fixpoint Systems in SBQL

SBQL provides querying capabilities similar to those of Datalog. The currently proposed solution is based on fixpoint systems, i.e. queries of the form $x = q(x)$, where x is a variable, q is an arbitrary SBQL query dependent on x. A system of such equations can have arbitrary number of variables. Such fixpoint systems in comparison to Datalog seem to have essential differences, in particular the following:

- Datalog is used to deduce *facts*, using other *facts* and *rules*. SBQL fixpoint systems are used to find objects or (complex) values which satisfy some conditions.
- Datalog is based on logic, thus in some authors expect that it would be possible to *prove mathematically* some properties of a Datalog program and its results.

SBQL theoretical foundations lie elsewhere, and the possibility of proving anything is not among the concerns of the SBQL design.

- The equations in SBQL fixpoint systems (which can be thought of as equivalent to Datalog rules) may use any valid SBQL query;
- SBQL puts no constraints on the negation operator and assumes neither stratification nor CWA. However, negation is not the only operation which may result in a query causing an infinite loop; for instance, another such operator is function *sinus* or arithmetic minus. SBQL assumes that the programmer takes appropriate care and such cases can be detected by testing.

In our opinion these differences concern mainly some specific rhetoric, ideological assumptions, terminology, and superficial notions. From the pragmatic point of view SBQL fixpoint systems are syntactically very similar to Datalog programs. Moreover, they can be used in the same situations and can solve the same tasks. For these reasons we consider SBQL fixpoint systems as a direct counterpart of Datalog programs. Taking in account all options, SBQL has the power of universal programming languages thus is incomparably more powerful than Datalog.

The syntax of an SBQL fixpoint system is as follows:

fixpoint(x_{i1}, x_{i2},..., x_{in}) {x_1 :- q_1; x_2 :- q_2;... x_m :- q_m;}

where:

- x_1, x_2,..., x_m are names of variables in this equation system,
- x_{i1}, x_{i2},..., x_{in} are returned variables, $\{x_{i1}, x_{i2},..., x_{in}\} \subseteq \{x_1, x_2,..., x_m\}$,
- q_1, q_2,..., q_m are SBQL queries with free variables x_1, x_2,..., x_m;

The semantics of this language construct is the following:

1. Variables x_1, x_2,..., x_m are initialized to empty bags.
2. Queries q_1, q_2,..., q_m are evaluated.
3. If the results of q_1, q_2,..., q_m are equal to the values of x_1, x_2,..., x_m, then stop (the fixpoint is reached). Otherwise assign the results of q_1, q_2,..., q_m to the values of x_1, x_2,..., x_m and go to step 2.
4. The values of x_{i1}, x_{i2},..., x_{in} are returned as the result of the fixpoint query.

As queries q_1, q_2,..., q_m can reference variables x_1, x_2,..., x_m, the fixpoint system provides recursive capabilities.

The simplest use of a fixpoint system in a query is the calculation of transitive closure. The query below uses a fixpoint system to find all subcomponents of the part named “engine” (the query addresses the schema shown in Fig.1):

```
fixpoint (parts){
  parts :- (Part where name="engine") union (parts.Component.leadsTo.Part);
}
```

Fixpoint systems are regular SBQL queries, and as such may be used as parts of other SBQL queries. The query below uses a fixpoint system as a part of a SBQL query, in order to find names of all unique engine elements:

```
distinct(fixpoint (parts){
  parts :- (Part where name="engine") union (parts.Component.leadsTo.Part);
}).parts.name
```

A fixpoint system may use some variables as a way to break down the problem into smaller, more manageable parts. The query below does that in order to calculate the number of different parts in the part named "engine":

```
fixpoint (final) {
  engine :- ((Part where name="engine") as x, 1 as howMany);
  engineParts :- engine union engineParts.Component.
                              ((leadsTo.Part) as x, (amount*howMany) as howMany);
  final :- (distinct(engineParts.x) as y).(y, sum(engineParts where x=y).howMany);
}
```

Only variable *final* is returned as the fixpoint result. The other two variables are used only to perform calculations, as their final values are inessential to the user. Variable *engine* is used to find the top element of the hierarchy (the "engine" part), while *engineParts* is the variable in which the results of recursive calculations are stored. Variables *final* and *engine* do not participate in the recursion.

The same principle is used in the next example. The query calculates the total cost and mass of the engine:

```
fixpoint (cost, mass){
  engine :- ((Part where name="engine") as x, 1 as howMany);
  engineParts :- engine union engineParts.Component.((leadsTo.Part) as x,
                                                     (amount*howMany) as howMany);

  engineDetails :- engineParts.((Detail) x as d, howMany as howMany);
  engineAggregates :- engineParts.((Aggregate) x as a, howMany as howMany);

  detailsMass :- sum((engineDetails.(howMany*d.Mass));
  detailsCost :- sum((engineDetails.(howMany*d.Cost));
  addedMass :- sum((engineAggregates.(howMany*a.assemblyMass));
  addedCost:- sum((engineAggregates.(howMany*a.assemblyCost));

  cost :- detailsCost + addedCost;
  mass :- detailsMass + addedMass;
}
```

Fixpoints, unlike transitive closures, are capable of evaluating more than one recursive problem in each step, in a manner similar to the Datalog. This topic may be an interesting area for further research, although most of the practical recursive problems we are aware of can be solved using only a single recursion.

Similarly to transitive closures, fixpoint systems may be used to perform recursive calculations without referring to the database. The example below shows a fixpoint system version of example calculating the square root of *a*:

```
fixpoint( x ){
  y :- (1 as r, 1 as c) union (y.(a/r + r)/2 as r, c+1 as c) where c ≤ 5;
  x :- (y where c = 5).r;
}
```

Fixpoint systems in SBQL fit well with the rest of the language. As they are based on a powerful and flexible approach, they are free from many drawbacks of Datalog, such as difficulty with complex objects, no inheritance, associations, methods, etc. When compared with transitive closures, fixpoint systems seem to be more readable, as decomposition of the problem is easier.

5 Recursive Procedures and Views in SBQL

SBQL philosophy allows for seamless integration of imperative language constructs with query operators, including recursive procedures and functions. This allows utilizing the most popular recursive processing technique, without sacrificing any of the benefits of query language. In contrast to popular programming languages the new quality of SBQL concerns types of parameters and types of functions output. The basic assumption is that parameters are any SBQL queries and the output from functional procedures is compatible with query output. Thus SBQL procedures and functions are fully and seamlessly integrated with SBQL queries. Parameters can be passed by *call-by-value*, *call-by-reference* and *strict-call-by-value* techniques.

Statements in SBQL procedures use SBQL queries. An SBQL query preceded by an imperative operator is a statement. Statements such as *if, while, for each*, etc. can be more complex than in typical programming languages, see [3]. SBQL includes many such imperative operators (object creation, assignment, insertion, deletion, flow control statements, loops, etc.).

Below we present a recursive procedure which finds all components of a specific part, along with their amount required to make this part. It consists of a single return statement. The returned value is an empty collection or the result from recursive invocation of the same procedure. For simplicity we omit types of parameters and the type of function result.

```
procedure SubPartsHowMany( myPartsHowMany ){
  return
    if not exists(myPartsHowMany) then bag()
    else bag( myPartsHowMany,
              SubPartsHowMany(myPartsHowMany.c.Component.
                              ((leadsTo.Part) as c, howMany * amount) as howMany))
            )
}
```

The procedure takes a collection of structures as the parameter (*myPartsHowMany*). Each structure contains *c* (a reference to a part) and *howMany* (the amount of parts).

The example below shows that this procedure may take a collection of parameters, instead of a single parameter, without any alterations:

```
SubPartsHowMany(bag((Part where name = "engine") as c, 68 as howMany),
                    (Part where name = "gearbox") as c, 135 as howMany)))
```

An advantage of recursive procedures is simplicity of the problem decomposition. A recursive task can be easily distributed among several procedures (some of which may be reused in other tasks). A procedure calculating the cost and mass of a part illustrating this possibility is shown below. The procedure utilizes the previously defined *SubPartsHowMany* procedure in order to perform the recursive processing and then performs calculations, on local variables (introduced by *create local*).

```
procedure CostAndMass(myPartsHowMany) {
  if not exists(myPartsHowMany) then return bag();
  create local SubPartsHowMany(myPartsHowMany) as parts;
  create local parts.((Detail) c as d, howMany as howMany) as details;
  create local parts.((Aggregate) c as a, howMany as howMany) as aggregates;
  create local sum(details.(howMany*d.detailMass)) as detailsMass;
  create local sum(details.(howMany*d.detailCost)) as detailsCost;
  create local sum(aggregates.(howMany*a.assemblyMass)) as addedMass;
  create local sum(aggregates.(howMany*a.assemblyCost)) as addedCost;
  return ((addedCost+detailsCost) as cost, (addedMass+detailsMass) as mass);
}
```

Recursive procedures in SBQL offer many advantages when compared to stored procedures in relational database systems. Most of them are consequences of the fact that procedures in SBQL are a natural extension of the SBA, working on the same principles and evaluated by the same evaluation engine, while in relational systems stored procedures are add-ons to the system evaluated separately from SQL queries. SBQL queries are valid as expressions, procedure parameters, etc. The type system is the same and there is no impedance mismatch between queries and programs.

SBQL updateable views are based on procedures and as such can be recursive and can utilize any other SBQL option, in particular parameters. Note that recursion without parameters makes little sense, thus if one assumes that views can be recursive then they must have parameters too. Recursive parameterized views are not available in any query language but SBQL. SBQL updateable views are discussed in detail in several publications, e.g. in [3, 5]. A simple read-only view, returning all subparts of parts which names are passed as a parameter, is shown below.

```
create view EnginePartsDef {
  virtual objects EngineParts (whichParts){
    if not exists(whichParts) then return bag();
    create local (Part where Name in whichParts) as p;
    return (p union EngineParts(p.Component.leadsTo.Part.name)) as b;
  }
  on retrieve do return b;
}
```

Here is an example invocation of this view:

EngineParts("pacer")

Similarly to transitive closures and fixpoint systems, recursive procedures may perform calculations without referring to the database. The example below shows a recursive procedure which calculates the square root of its parameter. The task is solved by two procedures, where the second one is recursive.

```
procedure sqrt( a ) {
  if (a < 0) then return -999999;     //exception
  return sqrtRecursive ( a, 1, 1 );
}

procedure sqrtRecursive ( a, x, c ) {
  if (c = 5) then return x;
  return sqrtRecursive ( a, (a/x + x)/2, c+1);
}
```

SBQL procedures and views can be used not only as a direct way of achieving the recursion via writing recursive procedures, but also as a means of writing reusable and parameterized queries with transitive closures and fixpoint systems. Here are the examples.

```
procedure sqrtTransitive(a){
  return   (  (1 as x, 1 as c)
              close by (((a/x + x)/2 as x, c +1 as c) where c ≤ 5))
           ).
           (x where c = 5)
}

procedure sqrtFixpoint(a){
  return
     fixpoint(x) {
        y :- (1 as r, 1 as c) union (y.(a/r + r)/2 as r, c+1 as c) where c ≤ 5;
        x :- (y where c = 5).r;
     }
}
```

6 Optimizations

A query language implementation without optimization is hardly accepted by the users due to bad performance. The amount of information stored in current databases would make the evaluation time of most queries unacceptable. The problem is even bigger in the case of recursive queries, as the evaluation cost of such queries is usually higher than in the case of non-recursive ones. It makes query optimization research a high priority task. Clearly defined semantics of SBQL allows for a systematic and disci-

plined approach to this problem. The adaptation of well-known techniques is possible. Query rewriting optimizations for SBQL are described e.g. in [3, 6]. The techniques useful for transitive closure queries are also presented there. Other optimization techniques, however, have not been investigated in detail yet. This applies to various index-based techniques, fixpoint system optimizations using semi-naïve evaluation and magic set techniques [7].

7 Recursive Queries – State of the Art

Currently three approaches to recursive query processing are prevalent:

- Extending SQL (or other query language) with the transitive closure operator;
- Languages based on deductive rules, such as Datalog; semantics of such languages can be expressed by fixed point equations;
- Utilization of stored procedures to provide recursive capabilities or delegation of recursive calculations to a universal programming language.

7.1 Transitive Closure

In relational databases introducing the transitive closure operator meets non-trivial problems:

- The operator cannot be expressed in the relational algebra, thus some extensions of the algebra have been proposed.
- The computational power of the transitive closure operator can be insufficient. Some advanced closures, e.g. the query "get the total cost and mass of a given part X" (assuming BOM structures) is impossible to express assuming typical syntax of the transitive closure operator [4].
- Calculation of transitive closure leads to performance problems. The issue has resulted in many algorithms, such as those described in [8, 9, 10].

SQL89 and SQL92 do not include transitive closure operator. Such operators are supported by Oracle and DB2 DBMS, and are included in SQL99 (aka SQL3) and SQL2003 standard proposals. In the following we present the solutions and discuss our test results for transitive closures implemented in Oracle and DB2.

Recursive Queries in Oracle. The Oracle DBMS provides support for recursive queries in the form of so called "hierarchical queries", with a CONNECT BY clause extending the SELECT statement. Queries can be further refined by a WHERE clause. Standard CONNECT BY clause returns an error if a loop is encountered in the queried data. Oracle 10g introduced a modified CONNECT BY clause, CONNECT BY NOCYCLE clause. It allows the programmer to formulate queries on data with cycles. Encountered cycles are ignored.

Oracle provides additional pseudocolumns, which can be used in queries, such as LEVEL (the level of a particular tuple in the hierarchy) or CONNECT_BY_ISLEAF

(is a particular tuple at the bottom of the hierarchy?). It also provides utility functions, which can be used in SELECT clause such as SYS_CONNECT_BY_ROOT which finds the root element of hierarchy for each selected element.

The Oracle solution to the problem of recursive queries suffers from serious drawbacks. First we note that from the semantic point of view it is very difficult to grasp, and it is additionally obscured a lot of proprietary options. Probably the most serious problem is the evaluation order of a query with a join. Materializing the join first might result in serious performance problems, when querying large tables. Unfortunately, query optimization by performing selections before joins (probably the most promising technique in this case) might be difficult due to the fact that the operation in question is not possible to express in the relational algebra. Any query optimization related to steps executed after the join operation will have a minor effect; probably it will not influence the cost of the most time consuming operations.

Auxiliary pseudocolumns are another sign of immaturity of the Oracle's solution. Oracle does not allow the users to perform any calculations in a recursive query (such as calculating the level of hierarchy a tuple is on). This makes such pseudocolumns necessary to formulate even the most basic queries. While this feature might be proven useful for a less experienced user (asking relatively simple queries), it is not as flexible as other approaches, e.g. like the approach represented by DB2 and the approach proposed for SBQL.

The effort put by the Oracle developers into maintaining the hierarchy of tuples in the result set does not seem to be justified. Users querying the database by any other means than the SQL console will have to reconstruct the hierarchy of elements anyway. In such a situation the approach presented by DB2 might be superior, because it allows the user to calculate information useful in recreation of the hierarchy during query evaluation.

Recursive Queries in DB2. Similarly to Oracle, DB2 supports recursive queries through transitive closures. The syntax and semantics of the corresponding solution is, however, totally different.

Unlike Oracle, DB2 does not provide pseudocolumns containing additional information. However, DB2 provides another powerful facility for programmers. It allows the programmers to perform their own calculations in recursive queries. Information, such as a tuple's level in the hierarchy, can be calculated according to the user's need. The tasks performed by functions in Oracle, such as finding root tuples or paths to a tuple (SYS_CONNECT_BY_ROOT and SYS_CONNECT_BY_PATH in Oracle), can be formulated in DB2 with no proprietary utility functions.

The solution for recursive queries in DB2 seems to be superior to the solution by Oracle. Instead of focusing on enhancing the language constructs by additional system functions and pseudocolumns, IBM has provided a flexible solution compatible as far as possible with the general SQL idea. Recursive queries in DB2 seem to be very similar to those proposed in SQL-99.

Recursive Queries in SQL-99. SQL-99 (aka SQL3) was the first SQL standard proposal that introduces recursive queries. Recursive queries in SQL3 are similar to

those implemented in DB2 (except for syntactic sugar). Their existence is their most important and useful feature. However, limiting to linear recursion is poorly justified. This imposes serious limitations of useful queries, as well as results in workarounds being used in relatively simple queries [11]. Also the restriction of operations which can be used in mutually recursive queries seem to be troublesome and poorly motivated. Because SQL3 does not have precisely defined semantics, it is impossible to accurately predict which queries will result in an oscillation. In effect, some queries, which would not result in oscillation, may be forbidden, while others, resulting in oscillation, are not. Another mechanism preventing execution of queries containing such infinite loops may be required.

7.2 Fixed Point Equations

Some recursive tasks cannot be expressed using transitive closure operation, but can be expressed using a fixed-point equation system. In order to solve a recursive task, the least fixed point of an equation system has to be found.

$$x_1 \leftarrow f_1(x_1, x_2, \ldots, x_n)$$
$$x_2 \leftarrow f_2(x_1, x_2, \ldots, x_n)$$
$$\ldots$$
$$x_n \leftarrow f_n(x_1, x_2, \ldots, x_n)$$

The fixpoint systems may be used in a similar way to transitive closures. The typical introductory example of a fixpoint system is a single fixpoint equation performing the computation of a transitional closure. However, as some professionals believe, the biggest potential of fixpoint systems may lie in evaluating the so-called *business rules*, i.e., the sets of many fixpoint equations used to solve complex recursive problems.

Fixed Point Equations in Datalog. Datalog is a database query language that some authors associate with artificial intelligence. Datalog is a manifestation of the paradigm known as *deductive databases* [12], claimed to be superior over typical databases due to some "intelligence" or "reasoning capabilities". The common terminology is that Datalog programs consist of *deductive rules*, where deduction is a form of strong formal reasoning with roots in mathematical logic. Because the rules can be recursive, their semantic model can be expressed by a least fixed point equation system. In particular, [13] presents the fixed point semantics of Datalog and [14] presents how semantics of Datalog can be expressed through fixed point equations over expressions of the relational algebra.

First papers on deductive databases appeared in 1983. Despite more than 20 years of history and very big pressure of academic community to introduce Datalog as a commonly used database query language (hundreds of papers, books, reports, dozens of academic projects, special conferences, journals, etc.) Datalog failed as a useful software production tool. According to our experience, the following disadvantages cause a catastrophic effect on the Datalog usability:

- Lack of efficient methodology supporting the developers of applications in transition from business conceptual models to Datalog programs. For real applica-

tions an average developer or programmer has no idea how to formulate Datalog rules in response to typical results of analysis and design processes (stated e.g. in UML).

- Although Datalog is claimed to be a universal query language, its real application area is very limited to niche applications requiring some "intelligence" expressed through syllogisms and recursive rules.
- Limits on data structures that Datalog deals with. Current object-oriented analysis and design methodologies as well as programming languages and environments deal with much more sophisticated data structures (e.g. complex objects with associations, classes, inheritance, etc.) than relational structures that Datalog deals with. Complex data structures allow one to get the software complexity under control.
- „Flatness" of Datalog programs, i.e. lack of abstraction and encapsulation mechanisms, such as procedures, views or classes. This flaw means no support for hierarchical decomposition of a big problem to sub-problems and no support for top-down program design and refinement and encapsulation of problem details.
- Datalog is stateless thus it gives no direct possibility to express data manipulation operations. The majority of applications require update operations, which are possible to express in Datalog only via side effects, with no clear formal semantics.
- Datalog implies significant problems with performance. Current optimization methods, such as magic sets, do not seem to be sufficiently mature and efficient.

Deductive Object-Oriented Databases. Datalog in its original form does not provide any facilities for processing complex data structures. Since late eighties researchers investigated the problem of combining object-oriented and deductive capabilities in a single DBMS. These efforts resulted in multiple implementations, reviewed in [15]. Three different strategies of the design of an OO deductive query language are possible:

- Language extension: existing language is extended with new (in this case OO-related) features.
- Language integration: a deductive query language is integrated with an imperative programming language, in the context of an object model or type system.
- Language reconstruction: a new logic language that includes object-oriented features is created, with an OO data model as a base.

Language reconstruction obviously requires more effort than the two other strategies, but it is likely to produce the best results. The language extension strategy may fail to capture all the aspects of OO programming and data model, as well as leads to detachment of the resulting query language from its theoretical foundations due to the introduction of features not originally intended to be a part of it. The success of the language integration strategy strongly depends on the degree to which the seamlessness of language integration is achieved.

One of the examples of OO deductive query languages is the OLOG query language described in [16]. OLOG is based on IQL, i.e. an older OO query language and the O2 object-oriented data model. It uses fixpoint semantics and syntax similar to Datalog, however it also supports data manipulation, which is a problem often overlooked in query languages (e.g. Datalog and OQL).

OLOG uses a database schema during query processing. The schema may contain *classes* and *relations*. Classes are collections of *objects* and relations are collections of *tuples*. The difference between an object and a tuple in OLOG is that the objects have unique object identifiers, and tuples do not, otherwise they're very similar, both can be nested and contain both complex and atomic values. OLOG classes support inheritance, including multiple inheritance. Unfortunately, as OLOG is only a pure query language. It is not integrated with programming language constructs and does not support procedures, functions or methods.

DOQL [17] is a deductive query language for ODMG-compliant databases. The language is an important contribution to the ODMG standard, as OQL (the primary query language for ODMG compliant OO databases) does not support recursive queries. DOQL does not differ much from OLOG when it comes to syntax and capabilities. It does, however, differ from OLOG in evaluation technique. DOQL queries are mapped to an object algebra, making the use of existing OQL optimization facilities possible (according to the claims of the authors).

Both OLOG and DOQL are typical examples of an OO deductive query languages. Derived from Datalog, they support a limited range of OO features (e.g. they support inheritance, but do not support method implementation) and utilize one of proven Datalog semantics (in this case fixpoint semantics). Their limitations, however, pose an important question: are they indeed object-oriented query languages, or just deductive query languages capable of processing complex data.

7.3 Recursive Procedures and Functions

Recursive procedures and functions are probably the most common approach to solving recursive problems. A recursive procedure contains direct or indirect call to itself. They can be used for efficient and elegant problem solution in a case when there is a solution for a very small scale (e.g. for an argument 1) and then there is a rule for expressing the problem on a bigger scale via solutions on the smaller scale (e.g. the solution for argument n is expressed via solutions for an argument $n-1$). Recursive functions are also commonly used for processing tree-like structures.

Recursive functions are among features of almost every programming language in common use. This applies also to the programming languages used to write stored procedures, such as the Oracle's PL/SQL and Transact SQL used in Microsoft SQL Server and Sybase databases, although most of those languages impose some limitations on the recursion. The semantics of recursive procedures requires an environment stack which is used to store parameters, local program entities and a return trace for each procedure invocation.

To be fully usable in the recursive queries context, recursive functions must be compatible with the domain of query outputs. Thus parameters for such functions

should be any queries, possibly returning any bulk output. Without explicit parameters recursive functions have little sense (they must be parameterized by side effects, which is a worse option). Moreover, the output from recursive functions should be compatible with query output too. Full orthogonality of language constructs requires that the domain of query output should be the same as the domain of function parameters and as the domain of function outputs. Unfortunately, this rather obvious requirement is not satisfied by recursive procedures and functions known from commercial systems.

Due to the limitations of database programming languages, recursive data processing is implemented outside the database management system within client applications. They perform the recursive processing on their own, utilizing facilities provided by programming languages such as Java or C++ to perform operations impossible or very difficult to implement in a particular database programming language. Such an approach, while attractive due to the ease of use and extensive standard libraries of modern programming languages, has also its drawbacks, serious enough to limit the usability. Main drawbacks are as follows:

- In contrast to database programming languages, database access is not seamlessly integrated with a programming language. The application programming languages have to use database access APIs, such as ODBC or JDBC.
- Database and programming language type systems are usually different. This results in impedance mismatch, i.e. the need to convert types during the processing.
- Usually parameters and output of programming language functions cannot be bulk, in contrast to results of queries. This means that in many cases recursion must be supported by iteration scanning sequentially bulk data. Such feature leads to clumsy code and problems with code writing and maintenance.
- Processing takes place outside DBMS, on the side of client application. It is more difficult to endanger the stability of DBMS with a poorly written function (for example one with an infinite recursion), and thus it reduces or eliminates the need for drastic security measures (such as the limit on depth of the recursion). At the same time, it seriously reduces performance due to the communication overhead and may generate large volume of network traffic, due to the high volume of transmitted data.

Those problems make recursive processing outside DBMS not always convenient. However, in most cases it is the only available choice.

7.4 State-of-the-Art Conclusions

Presented recursive query facilities are the most widespread and well known solutions of the recursive query problem. Unfortunately, they are also representative for the group – available solutions do not differ much from those presented.

Recursive query processing, despite the interest in the problem shown by both academic and commercial communities, is still immature. Most of the current research focuses on removing the inadequacies of existing solutions, but unfortunately, the

chance of radically improving the situation is small, as the inadequacies come from the foundations of a particular approach.

The transitive closure is a concept beyond the relational algebra on which SQL is based on. Introducing this feature undermines optimization techniques such as query rewriting. Recursion causes that it is much more difficult to guarantee that the rewritten query will be semantically equivalent. Most of the optimization techniques for transitive closure queries are based on workarounds such as materialized views (cached query results), which introduce new problems (like the maintenance of materialized views).

Fixpoint equation systems have much potential. Unfortunately, Datalog (the only wide known language, which semantics can be described in terms of fixpoint equation systems) is seriously flawed as a general purpose query language. It does not cover the problem of imperative operations, such as updates, deletes etc. The Datalog variants based on the first-order logic are limited in their expressiveness. More expressive variations do not have such well understood theoretical foundations, which makes optimization (and providing any proofs to support the "deductions") more difficult. It also does not take into account user needs. The most popular queries on values of tuple attributes are not well supported by Datalog. It mostly concentrates on "deducing" facts using other facts and rules. All those drawbacks, as well as the trend of presenting the Datalog using formalized mathematical language (not necessarily liked or well understood by most programmers), result in the effect that Datalog is still not accepted by the commercial world. Object-oriented deductive query languages, based on Datalog usually utilize fixpoint semantics. However, they usually do not provide the full range of features required in an useful OO query language. Thus they may be interesting as prototypes and the basis for further research, but their potential for practical applications is currently very limited.

Recursive functions as a way to implement recursive queries also have drawbacks. Even in database programming languages, such as PL/SQL, the binding to the database is not exactly seamless. For example, it is still impossible to return a tuple (or a set of tuples) as the result of a function. The standard libraries provided lack of some very important features (such as collections). Recursive functions in their current form are also unsuited to ad-hoc querying.

The discussion presented above shows immaturity of current solutions to the problem of processing recursive queries. Those problems are the result of limited conceptual foundations on which those solutions are based. The only possible way to get rid of those problems is using a completely different foundation for creating a mature and consistent solution.

8 Conclusions

We have presented recursive query processing capabilities for the Stack-Based Query Language (SBQL). SBQL offers very powerful and flexible recursive querying facilities for object-oriented environments. Within a universal framework SBQL provides

both object-oriented (classes, inheritance) and deductive (transitive closure, fixpoint) features.

The transitive closure allows formulating queries more powerful and easily readable than SQL queries when compared with Oracle and DB2 SQL variants of transitive closure operators. Combined with the ease of semi-structured data handling in SBQL this may make XML data processing much easier.

Fixpoint systems provide SBQL with recursive capabilities similar to deductive query languages. However SQBL offers much more freedom, as there are no restrictions on operators which may be used within the queries. SBQL is also much better prepared to handle structured and semi-structured data than Datalog and its variants. This freedom, however comes at a cost, because the programmer must make sure that the query does not start an infinite evaluation loop.

Recursive procedures and views provided by SBQL allow to solve easily complex problems through problem decomposition, code reuse and other facilities typical for imperative programming languages. They are seamlessly integrated with the querying capabilities and allow the programmer to fully benefit from all the query language and DBMS properties, i.e. macroscopic statements, handling of bulk data, persistent storage and optimization for queries used within procedures.

With the recent rise of interest in recursive processing due to the emergence of XML, RDF and other similar standards the SBQL seems to provide an interesting and universal alternative to other query languages.

References

1. K.Subieta, C.Beeri, F.Matthes, J.W.Schmidt. A Stack-Based Approach to Query Languages. Proc. East-West Database Workshop, 1994, Springer Workshops in Computing (1995)
2. K.Subieta, Y.Kambayashi, and J.Leszczyłowski. Procedures in Object-Oriented Query Languages. Proc. VLDB Conf., 182-193, Morgan Kaufmann (1995)
3. K.Subieta. Theory and Construction of Object-Oriented Query Languages. Editors of Polish-Japanese Institute of Information Technology, Warsaw 2004, 522 pages, (in Polish)
4. M.P.Atkinson, P.Buneman. Types and Persistence in Database Programming Languages. ACM Computing Surveys 19(2):105-190, ACM (1987)
5. H.Kozankiewicz, K.Subieta. SBQL Views –Prototype of Updateable Views. Proc. 8th East-European Conference on Advances in Databases and Information Systems (ADBIS), September 2004, Budapest, Hungary.
6. J.Płodzień, K.Subieta. Applying Low-Level Query Optimization Techniques by Rewriting, Proc. DEXA Conf., Lecture Notes in Computer Science 2113: 867-876, Springer (2001)
7. J.D.Ullman. Principles of Database and Knowledge-Base Systems, volume II, ch. 13, W H Freeman, 1990
8. F.Fotouhi, A.Johnson, S.P.Rana. A hash-based approach for computing the transitive closure of database relations, The Computer Journal vol. 35 no. 3: A251--A259, Oxford University Press (1992)
9. W.Yan, N.M.Mattos. Transitive Closure and the {LOGA} + -Strategy for its Efficient Evaluation, Mathematical Fundamentals of Database Systems, Lecture Notes in Computer Science 364: 415-428, Springer (1989)

10.S.Taylor, N.I.Hachem, A Direct Algorithm for Computing the Transitive Closure of a Two-Dimensionally Structured File, Lecture Notes in Computer Science 495: 146-159, Springer (1991)
11.J.D.Ullman, J.Widom. A First Course in Database Systems, ch.5, Prentice Hall, 1997
12.H.Gallaire, J.Minker, J.-M.Nicolas: Logic and Databases: A Deductive Approach. ACM Comput. Surv. 16(2): 153-185, ACM (1984)
13.S.Abiteboul, R.Hull, V.Vianu. Foundations of Databases, ch. 12, Addison-Wesley (1995)
14.S.Ceri, G.Gottlob, L.Tanca. What You Always Wanted to Know About Datalog (And Never Dared to Ask). IEEE Transactions on Knowledge and Data Engineering 1(1): 146-167, IEEE Computer Society (1989)
15.P.R. Falcone Sampaio, N.W. Paton Deductive Object-Oriented Database Systems: A Survey, LNCS: Proceedings of the Third International Workshop on Rules in Database Systems: 1 - 19, Springer-Verlag (1997)
16.X. Li and M. Liu, Design and Implementation of the OLOG Deductive Object-Oriented Database System. In Proceedings of the 11th International Conference on Database and Expert Systems Applications (DEXA 2000), London, Greenwich, UK. September 4-8, 2000. Lecture Notes in Computer Science, Vol. 1873, Springer 2000
17.P. R. F. Sampaio, N. W. Paton, Deductive Queries in ODMG Databases: the DOQL Approach, Proceedings of the 5th International Conference on Object-Oriented Information Systems (OOIS): 57-74, Springer-Verlag (1998)