Tomasz Pieciukiewicz, Kazimierz Subieta

Recursive Query Processing in SBQL

Nr 979

Warszawa, November 2004

Abstract Recursive queries are required for many tasks of 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 of theoretical foundations. In particular, this concerns corresponding extensions of SQL in Oracle and DB2 systems. In this report we present recursive query processing capabilities for the object-oriented Stack-Based Query Language (SBQL) and compare them with similar capabilities in variants of SQL. SBQL offers very powerful and flexible recursive querying capabilities due to the fact that recursive processing operators are fully orthogonal to other features of this language. This report specifies corresponding SBQL constructs, such as transitive closures and fixed point equations. We compare them to other query languages, in particular to Datalog. We also present briefly optimization possibilities for recursive queries.

Keywords: Stack-Based Approach (SBA), query languages, SBQL, object-orientedness, transitive closures, fixed-point equations, deductive rules, recursive functions, recursive views

---

Streszczenie Rekurencyjne zapytania są wymagane dla wielu zadań występujących w rzeczywistych aplikacjach baz danych. Wśród nich możemy wymienić rachunek materiałów (BOM), różne typy sieci (transportowa, telekomunikacyjna, itp.), przetwarzanie danych półstrukturalnych (XML, RDF) itd. Wspomaganie dla rekurencyjnych zapytań w obecnych językach zapytań jest ograniczone i pozbawione podstaw teoretycznych. W szczególności, dotyczy to odpowiednich rozszerzeń SQL w systemach Oracle i DB2. W raporcie przedstawiamy możliwości w zakresie rekurencyjnych zapytań w języku zorientowanym obiektowo Stack-Based Query Language (SBQL) i porównujemy je do podobnych możliwości w wariantach SQL. SBQL oferuje bardzo mocne i elastyczne możliwości w zakresie rekurencyjnych zapytań dzięki temu, że operatory służące do przetwarzania rekurencyjnego są całkowicie ortogonalne w stosunku do innych cech języka. Raport specyfikuje odpowiednie konstrukcje SBQL, takie jak tranzytywne domknięcia oraz równania stałopunktowe. Porównujemy je do innych języków zapytań, w szczególności do Datalogu. Na zakończenie prezentujemy krótko możliwości optymalizacji zapytań rekurencyjnych.

Słowa kluczowe: podejście stosowe, SBA, języki zapytań, SBQL, obiektowość, tranzytywne domknięcia, równania stałopunktowe, reguły dedukcyjne, rekurencyjne funkcje, rekurencyjne perspektywy.

Content

1 Introduction

There are many important tasks that require recursive processing. The most widely known is Bill-Of-Material (BOM), 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, e.g. cars or airplanes. Typical BOM software processes such data structures by proprietary routines and applications implemented in a classical programming language. Frequently, however, there is a need to use *ad hoc* queries or programs addressing such structures. In such cases the user requires special user-friendly facilities dealing with recursion in a query language. Similar problems concern processing and optimization tasks on genealogic trees, stock market dependencies, various types of networks (transportation, telecommunication, electricity, gas, water, and so on), etc. Recursion is also necessary for internal purposes of computer systems, such as processing recursive metadata structures (e.g. Interface Repository of the CORBA standard), software configuration management repositories, hierarchical structures of XML or RDF files, and others.

In many cases recursion can be substituted by iteration, but this implies much lower programming level and less elegant problem specification. Iteration may also cause higher cost of program maintenance since it implies a clumsy code that is more difficult to debug and change.

Despite of its importance, recursion is not supported in SQL standards (SQL-89 and SQL-92). Beyond the standards, it is implemented (differently) in relational DBMSs, in particular, in Oracle and DB2, in the form of transitive closures. Newer SQL standards known as SQL-99 (aka SQL-3) and SQL 2003 introduce both transitive closure operator and deductive rules *a la* Datalog. Unfortunately these standards are very huge and eclectic thus there are doubts among database professionals if they will ever be fully implemented, especially concerning such advanced properties. Recursion is also considered a desirable feature of XML-oriented and RDF-oriented query languages but current proposals and implementations do not introduce such a feature or introduce it with many limitations.

The ODMG standard and its query language OQL do not mention any facilities for recursive processing. This will obviously lead to clumsy codes when such tasks have to be programmed in programming languages such as C++, Java and Smalltalk.

The possibility of recursive processing has been highlighted in the deductive databases paradigm, 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), poor performance and perhaps others. Thus Datalog applications supporting all the scale that databases require are till now unknown. Nevertheless, the idea of Datalog semantics based on fixed-point equations seems to be very attractive to formulate complex recursive tasks. Note however that fixed-point equations can be introduced not only to languages based on logic programming, but to any query language, including SQL, OQL and XQuery.

Besides transitive closures and fixed-point equations there are classical methods 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 that 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 possess parameters, which could be bulk types compatible with query output too. Currently very few existing query languages have such possibilities, thus using recursive functions or views in a query language is practically unexplored.

This report discusses three different approaches to recursive querying:

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

The first approach is implemented in a few industrial commercial systems, as will be shown, with severe limitations. Two next approaches are implemented so far in research prototype systems, with limitations too. Till now no system implements all three approaches in a sufficiently robust version, thus comparison of them from the software designer and programmer point of view is unfeasible. Moreover, all widely known implementations concerns relational database systems which time has passed. There are few theoretical proposals concerning recursive queries for object-oriented and XML-oriented systems, but implementations are not widely known or very limited.

In this report we describe all three approaches to recursive processing within the unified framework - the Stack Based Approach (SBA) to object-oriented query/programming languages. SBA introduces an own query language Stack-Based Query Language (SBQL) based on abstract, compositional syntax and formal operational semantics (abstract implementation). Within this approach all three approaches are or 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 report we compare the approaches on sufficiently complex examples showing strengths and weakness of them with respect to some situations that appear in database application programming.

Recursive queries can imply performance problems. There are several typical optimization methods related to recursive queries, such as factoring out independent subqueries, indices, or caching (materializing) results of queries to utilize them during evaluation of next (identical) queries. For fixpoint equations there are special methods know as magic sets. In the report we briefly discuss the above issues.

The rest of the report is organized as follows. In Section 2 we present the widely known solutions of recursive processing in popular relational DBMS and in Datalog. Section 3 is a short introduction to the SBA and SBQL. Next sections present facilities for recursive processing that are or can be introduced to SBQL: transitive closures (Section 4), fixpoint equations (Section 5) and recursive procedures (Section 6). In these sections we compare these facilities on some real examples and discuss their advantages and disadvantages. In Section 7 we shortly discuss optimization opportunities for these methods, in particular, rewriting techniques for transitive closures and magic sets for fixpoint equations. Section 8 concludes.

2 Recursive Query Processing - the 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;

Several other, less popular approaches to this problem will also be discussed.

2.1 Transitive Closure

The transitive closure of a binary relation R is the smallest transitive relation R* which includes R, i.e.

$x \; R \; y \Rightarrow x \; R^* \; y$

$x \; R \; y \wedge y \; R^* \; z \Rightarrow x \; R^* \; z$

Many real life problems can be reduced to a transitive closure problem, for instance, algorithms operating on genealogical trees, processing BOM data structures, operations on various kinds of networks, etc.

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 transitive closure operator syntax [AB87].

- Calculation of transitive closure leads to performance problems. The issue has resulted in many algorithms, such as those described in [FJR92], [YM89] and [TH91].

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 DBMS-s.

2.1.1 Recursive Queries in Oracle

The Oracle DBMS provides support for recursive queries in the form of so called “hierarchical queries”. Their syntax is as follows:

SELECT selected data
FROM table_name **CONNECT BY** condition
[**START WITH** entry point]

- *selected data* is specified as in normal SELECT statement;
- *table_name* specifies the name of one or more tables. In Oracle 8i and earlier versions, only a single table or a view could be used in a recursive query. Starting with Oracle 9i multiple tables are also supported;
- *condition* specifies the condition under which relation between two tuples occurs. A part of the condition has to use the PRIOR operator to refer to the parent tuple. The part of the condition containing the PRIOR operator must have one of the following forms:
- **PRIOR** *expr* *comparison_operator* *expr*
expr comparison_operator **PRIOR** *expr*
- where *expr* is a standard SQL expression referring to one or more columns, *comparison_operator* is any of SQL operators;
- *entry point* specifies the conditions for the starting set of tuples. Specification of an entry point is optional. If it is not specified, all tuples in the table will be used as the entry point;

Query samples below, as well as the query samples for DB2 use the database structure from Fig.1. PK denotes primary key, FK - foreign key. Each department, except the highest in the departments’ hierarchy, contains the key of its directly superior department as the value of attribute *superdept*.

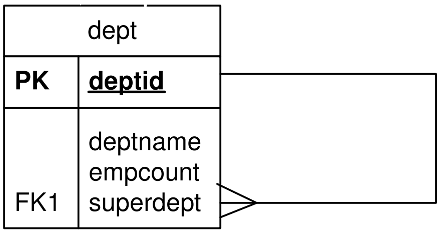

Fig. 1 Sample database structure

The following example query returns the number of employees in the department called “Production”, including all its subdepartments:

Example 1	**SELECT** sum(empcount) **FROM** departments **CONNECT BY** superdept = **PRIOR** deptid **START WITH** deptname = 'Production';

Query can be further refined by a WHERE clause. The query is evaluated in the following order:

1. Any joins, i.e. tables listed after FROM together with join predicates after the corresponding WHERE, are materialized first;
2. The CONNECT BY processing is applied to the rows returned by the join operation. The rows specified by START WITH are used in the first iteration and rows returned by previous iteration in the further iterations.
3. Any filtering predicates from the WHERE clause (those not used in the join operation) are applied to the results of the CONNECT BY operation.

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. The pseudocolumns are:

- LEVEL - indicating the level of a particular tuple in the hierarchy;
- CONNECT_BY_ISLEAF - indicating, whether a particular tuple is at the bottom of the hierarchy;
- CONNECT_BY_ISCYCLE – indicating, whether a particular tuple introduces a cycle into the hierarchy;

Oracle also provides utility functions, which can be used in SELECT clause such as:

- SYS_CONNECT_BY_ROOT – finds the root element of hierarchy, for each selected element;
- SYS_CONNECT_BY_PATH – finds the path (or paths) from the root elements to the selected elements;

In order to allow ordering of selected elements within the hierarchy, an ORDER BY SIBLINGS clause has been introduced. This clause separately orders child elements of each element in the hierarchy, according to the given ordering rules.

Oracle uses a search-depth-first algorithm for querying the hierarchy. This means that each tuple in the result set is first followed by tuples, which are its children. Its siblings (tuples on the same level in the hierarchy) are listed after its children. For example, we fill in the table *departments* depicted on the Fig.1 by values presented in Table 1 and then formulate some queries.

deptid	deptname	empcount	superdept
1	Marketing	30	NULL
2	Production	10	NULL
3	Engine assembly	15	2
4	Chassis assembly	12	2
5	Final assembly	22	2
6	Superstructure assembly	12	4
7	Door assembly	6	4
8	Locks assembly	9	7

Table 1. Data for Example 2

Consider the following query:

Example 2	**SELECT RPAD**('', level-1, '--') || deptname **FROM** departments **CONNECT BY** superdept = PRIOR deptid **START WITH** deptname = 'Production';

RPAD is a formatting function provided by Oracle. It is used to format output in order to show element hierarchy. The query returns the following result set:

Example 2 – result set	Production --Engine assembly --Chassis assembly

	----Superstructure assembly ----Door assembly ------Locks assembly --Final assembly

The Oracle solution to the recursive query problem 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 predicate. Materializing the join first might result in serious performance problems, when querying large tables. Unfortunately, query optimization by performing selection before join (probably the most promising technique in this case) might be difficult due to weakness of SQL theoretical foundations – especially, since 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 (such as the ORDER BY SIBLINGS clause) 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.

2.1.2 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. A typical recursive query in DB2 consists of four parts:

- A virtual table in the form of a common table expression. The virtual table will be used to hold the results of the query;
- A query used to establish the initial result set (and populate the virtual table with data on the top of the hierarchy);
- Recursively called query, added to the virtual table through the operator UNION ALL ;
- A final SELECT statement that yields the required rows from the output;

The query syntax is as follows:

```
WITH virtual_table_name (virtual_table_columns) AS
(
SELECT ROOT.parent_ID, ROOT.item_ID, ROOT.items
FROM queried_table ROOT
[WHERE condition]
UNION ALL
SELECT CHILD.parent_ID, CHILD.item_ID, CHILD.items
FROM virtual_table_name PARENT, queried_table CHILD
WHERE PARENT.item_ID = CHILD.parent_ID
)
SELECT virtual_table_columns
FROM virtual_table_name
```

The meaning of the query components is the following:

- *virtual_table_name* is the name of virtual table storing the result of recursive query;
- *virtual_table_columns* are the column names in the virtual table;
- *parent_ID* is the column name with the foreign key of the parent element;
- *item_ID* is the name of column with primary key of an element;
- *items* are names of selected columns;
- *queried_table* is the name of the queried table
- *condition* specifies condition that have to be met by a tuple to be considered as a root element in the hierarchy;
- *ROOT, CHILD, PARENT* are names assigned to tables (ROOT and CHILD to the queried table, PARENT to virtual table) in order to make the queries unambiguous;

The following sample query returns the number of employees in department called "Production", including all its subdepartments (same as the sample Oracle query):

Example 3

```
WITH temptab(deptid, empcount, superdept) AS
(
SELECT root.deptid, root.empcount, root.superdept
FROM departments root
WHERE deptname='Production'
UNION ALL
        SELECT sub.deptid, sub.empcount, sub.superdept
        FROM departments sub, temptab super
        WHERE sub.superdept = super.deptid
)
SELECT sum(empcount) FROM temptab
```

In DB2, the evaluation order of a recursive query is as follows:

- A virtual table is initialized:

  WITH temptab(deptid, empcount, superdept)

- The initial result set is estabilished in the virtual temporary table:

```
SELECT root.deptid, root.empcount, root.superdept
FROM departments root
WHERE deptname='Production'
```

- The recursion takes place, by joining each tuple in the temporary table with the child tuples, until the result set stops growing:

```
SELECT sub.deptid, sub.empcount, sub.superdept
FROM departments sub, temptab super
WHERE sub.superdept = super.deptid
```

- The final query extracts the requested information from the temporary table:

```
SELECT sum(empcount) FROM temptab
```

Unlike Oracle, DB2 does not provide pseudocolums 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.

DB2 uses a search-width-first algorithm for querying the hierarchy. This means that in the natural order of tuples in the result set we get the tuples on the top of the hierarchy first; then, all the tuples immediately below the top etc.

Example 4: For the table *departments* shown in Fig.1 and Table 1 consider the following query:

Example 4	**WITH** temptab(superdept, deptid, deptname, empcount, level) **AS** (**SELECT** root. superdept, root. deptid, root.deptname, root.empcount, 1 **FROM** departments root **WHERE** superdept is null **UNION ALL** **SELECT** sub.superdept, sub.deptid, sub.deptname,sub.empcount, super.level+1 **FROM** departments sub, temptab super **WHERE** sub.superdept = super.deptid

	) **SELECT** **VARCHAR**(**REPEAT**('--', level-1) || deptname , 60) **FROM** temptab;

The query returns the following result set:

Example 4 – the result set	Production --Engine assembly --Chassis assembly --Final assembly ----Superstructure assembly ----Door assembly ------Locks assembly

The query uses the facility provided by DB2 to calculate tuple's level in the hierarchy. In the initial result set the "level" column is assigned value 1. In each step of the recursion, tuples newly added to the result set have the "level" column set to a value equal to the "level" of their parents increased by 1. 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.

2.1.3 Recursive Queries in SQL-99

SQL-99 (aka SQL3) was the first SQL standard proposal that introduces recursive queries. Similarly to DB2, a typical recursive query in SQL-99 consists of four parts:

- WITH keyword;
- One or more relation definitions. The definitions are separated by commas, each consists of the following elements:
  - Optional keyword RECURSIVE, if defined relation is recursive;
  - Name of defined relation and names of columns in virtual table storing the relation (*table_definition* below);
  - AS keyword;
  - Query defining the relation (*initial_query* below);
  - If the relation is recursive, a query defining recursion in the relation;

- Query, which can use any and all of defined virtual tables to yield the required rows (*final_query* below);

  The query syntax is as follows:

```
WITH [RECURSIVE] table_definition AS initial_query [UNION recursive query]
final_query
```

Queries defining recursive relations are subject to some limitations:

- Recursion has to be **linear**. This means that the FROM clause of the select statement may include only a single occurrence of a recursive relation (for example, if a relation dependent is recursive, the clause FROM dependent AS r1, dependent AS r2 is incorrect);
- Negation of mutually recursive relations has to be **stratified**. This means that dependency graph for the query cannot contain a cycle in which one of the graph edges represents negation operation. The dependency graph is constructed as follows:
  - Vertices of the graph represent relations;
  - If a relation contains negated reference to another relation, create an edge between vertices representing those two relations, marked by a symbol "-";
  - If a relation contains a (not negated) reference to another relation, create an edge between vertices representing those two relations. Do not mark the edge.

If the graph contains a cycle in which one or more edges are marked with symbol "-", the negation is not stratified, and such query is not a valid SQL3 query. Similar restrictions apply to operations other than negation, if their use may result in a query that is non-monotone such as aggregation. Non-monotonic queries result in queries that do not converge, but fall into oscillation instead – tuples are added to and removed from the result set in an infinite loop.

Recursive queries in SQL3 are similar to those implemented in DB2 (except for syntactic sugar). Their introduction is in the authors' opinion one of it's most important and useful features. However, limiting the users to linear recursion is not justified, while imposing serious limits on users, as well as resulting in workarounds being used in relatively simple queries [UW97]. Also the limitations on operations, which can be used in mutually recursive queries seem to be imperfect. Because SQL3 does not have precisely defined semantics, it is impossible to accurately predict which queries will result in oscillation. Some queries, which would not result in oscillation, may be forbidden, while others (resulting in oscillation) won't. Another mechanism preventing execution of queries containing such infinite loops may be required.

2.1.4 Other transitive closure implementations

Transitive closure has been introduced not only as a feature in commercially sold DBMSs, but also as a research tools in prototype implementations. One of such proposals is presented in [DM92]. It adds the possibility to use recursive views within SQL queries.

The recursive queries are defined similarly to non-recursive queries in the standard SQL:

CREATE VIEW *viewname* (*attributed-column-list*)

AS [*set-type*] **FIXPOINT**

OF *table-name* [(*column-list*)]

BY SELECT *select-list*

FROM *table-reference*, *view-reference*

where-clause;

The view definition is very similar to the standard SQL view definition, with exceptions described below:

- The attributed-column-list extends the standard column-list of SQL. With **INC** and **DEC** respectively, the specification of monotonous characteristics of certain attributes is allowed. This information is crucial in optimization and assuring the finiteness of certain queries;
- The *set-type* (**ALL** or **DISTINCT)** specifies, whether duplicates should be eliminated, or duplicate tuples must be taken into account;
- *table-reference* should point to the source table, which provides data for the view;
- *view-reference* should point to the view being currently defined;
- *where-clause* should describe transitive closure condition.

Although from the syntactic point of view the changes seem to be small they offer a serious challenge when it comes to implementation. Any recursive query capability has to be as closely as possible integrated with the query evaluation engine, otherwise the performance will be very poor. The creators of this extension did not create a new DBMS, creating a front-end to an existing DBMS instead. They claim the results and choice of architectural variant as "adequate", but unfortunately, do not present any performance measurements.

2.1.5 Caching Transitive Closure Query Results

Transitive closure queries on relational databases are expensive. The cost analysis of transitive closure queries can be found in [Li93]. If such queries are performed often and the data is not updated frequently a cached query result can be used as a way of avoiding the high cost of evaluation.

Cached query results have to be properly maintained in case of database updates. When a database update is performed its effect on a cached query result has to be determined and appropriate changes to the underlying derived data structures have to be made. Otherwise, to keep consistency, it would be necessary to reevaluate the entire results after each update. This process is called incremental evaluation and in case of cached results of transitive closure queries the issue is not trivial.

Incremental evaluation systems (IES) have been described in [Do99]. In [Li97] the power of IES has been examined in detail, especially the differences between practical relational database systems and pure relational calculus in maintaining queries containing nested relations (or their representation using flat relations and auxiliary tables) and aggregate functions. The problem of cached query result maintenance is also covered in [Do97], with an in-depth discussion of properties of query languages that make cached query results non-maintainable in relational calculus and facilities required in order to maintain such results.

Cached transitive closure query results may be seen as useful tools when dealing with databases without efficient facilities for ad-hoc transitive closure queries. They may also be considered a useful tool for performance optimization in databases which provide such facilities, assuming they are not updated frequently but extensively queried. In general, however, the query optimization problem for the general case of cached transitive closure query results seems to be very hard by its inherent nature; thus we do not expect here any silver bullet.

2.2 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:

$$
\begin{aligned}
x_1 &\;\text{<-}\; f_1(x_1, x_2, \ldots, x_n) \\
x_2 &\;\text{<-}\; f_2(x_1, x_2, \ldots, x_n) \\
&\ldots \\
x_n &\;\text{<-}\; f_n(x_1, x_2, \ldots, x_n)
\end{aligned}
$$

has to be found.

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.

2.2.1 Fixed Point Equations in Datalog

Datalog is a database query language that some authors associate with the artificial intelligence. Datalog is a manifestation of the paradigm known as *deductive databases* [GMN84], claimed to be superior over typical databases due to some extra "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, [AHV95] presents the fixed point semantics of Datalog and [CGT89] presents how semantics of Datalog can be expressed through fixed point equations over expressions of the relational algebra.

A Datalog program is composed from a set of *rules* and *facts*. Facts are assertions about a relevant piece of the world, such as "John is parent of Mary". Rules are sentences that allow us to deduce facts from other facts, for example "if X is a parent of Y and Y is a parent of Z, then X is a grandparent of Z". Rules, in order to be general have to contain *variables* (in our example X, Y and Z). In Datalog both facts and rules are represented as Horn clauses:

$$L_0 \;\mathbf{:\text{-}}\; L_1, L_2, \ldots , L_n$$

where each L_i is a *literal* of the form $p_i(t_1, t_2, \ldots, t_k)$ such that p_i is a *predicate symbol* and the t_j are *terms*. Each *term* is either a *constant* or a *variable*. The left hand side of a Datalog clause is called *head*, the right hand side is called *body*. Facts have empty bodies; clauses with at least one literal in the body represent rules.

A sample Datalog fact, stating that John is a parent of Mary is shown below:

Example 5	parent(John,Mary)

and a sample Datalog rule, stating that "if X is a parent of Y and Y is a parent of Z, then X is a grandparent of Z" might look like this:

Example 6	grandparent(X,Z) **:-** parent(X,Y),parent(Y,Z)

Predicates available to a user depend on a particular Datalog variation. The standard Datalog (called also *pure Datalog*) does not allow predicates, except for those defined in a Datalog program, to be used in clauses. The pure Datalog does not allow negation to be used either. Both would endanger the *safety* of Datalog programs. Safety means that any Datalog program should have a finite output.

Predicates defined as rules in the pure Datalog programs correspond to *finite* relations, unlike predicates represented by symbols like "<" or "≠", which correspond to *infinite* relations. Built-in predicates may be considered from the formal point of view as normal predicates with different physical realization – for instance, as predefined procedures instead of definition within a Datalog program. Each variable occurring as an argument of a built-in predicate in a rule body must also occur in an ordinary (non built-in) predicate of the same rule body, or be bound by an equality (or sequence of equalities) to such variable or an constant. This allows the user to introduce such predicates, without introducing infinite relations. Introducing built-in predicates allows the user for more flexible and easy querying. Without such predicates each relation (like "<") has to be explicitly defined by programmer for each constant value.

Negation is another problem in Datalog queries, as it also can introduce infinite relations and oscillation during evaluation of a Datalog program. The problem of infinite relations that can be introduced by negation can be solved by the Closed World Assumption (CWA). CWA in the Datalog case can be formulated as follows:

If a fact does not logically follow from a set of Datalog clauses, then we conclude that the negation of this fact is true.

This assumption applied to the pure Datalog allows deducing negative facts from a set of Datalog clauses, however it does not allow to use these negative facts to deduce further facts. This means that using the pure Datalog to express rules like "if X is Y and X is not Z, then X is A" (for example, "if X is a student and X is not graduate student, then X is an undergraduate student") is impossible.

An extension of Datalog, called Datalog¬, allows using negated literals in rule bodies. For the safety reasons it is required that each variable occurring in a negated literal also appears in a non-negated literal in the same rule body. A sample rule in Datalog¬, defining the properties of undergraduate students could be formulated as follows:

Example 7	und(X) **:-** stud(X), ¬grad(X)

(the predicate symbol *und* represents undergraduate students, *stud* represents students, *grad* represents graduate students).

Datalog¬ evaluation is not an easy problem. Datalog¬ programs may have more than one minimal set of facts (so-called minimal Herbrand model). For instance, program

Example 8	boring(chess) **:-** ¬interesting(chess)

has two minimal sets of facts satisfying the rules: {boring(chess)} and {interesting(chess)}. An important question is which one of the minimal models should be chosen?

The most common approach to this problem is presented by the Stratified Datalog¬. It uses a specific evaluation policy, called *stratified evaluation*. When evaluating a rule with one or more negative literals in the body, predicates corresponding to these negative literals are evaluated first, then the Closed World Assumption is applied "locally" to these predicates.

In our example, we would evaluate the predicate *interesting* first. Since there are no rules or facts that would allow us to deduce any fact of the form *interesting(X)*, the set of positive answers to this predicate is empty, in particular *interesting(chess)* cannot be derived. By applying CWA "locally" to the *interesting predicate*, we derive *¬interesting(chess)*. Now we evaluate the rule and get *boring(chess)*. Thus, the minimal model is {boring(chess)}.

When a Datalog¬ program consists of several rules, evaluation of one rule may require evaluation of further rules and so on. Each of these rules may contain negated literals in their bodies. It is required that it is always possible to completely evaluate all the negated predicates which occur in the rule body, or the bodies of subsequent rules (including all the predicates, which are required to evaluate the negated predicates) before evaluating the rule head. A program, which fulfills this condition, is called *stratified*.

Another evaluation paradigm for Datalog¬ programs is called *inflationary evaluation*. Unlike *stratified evaluation*, it applies to all Datalog¬ programs. Inflationary evaluation is performed iteratively. All rules in the program are processed in parallel in each step. At each step the CWA is used during the evaluation of rule bodies – it is assumed that the negation of all facts not yet derived is valid. The evaluation is finished when no more additional facts can be derived. Unlike stratified evaluation, inflationary evaluation does not always produce a minimal model for the given program.

Datalog extensions, such as negation and built-in predicates are important steps toward making Datalog more useful as a query language. Without them Datalog can only be seen as an academic language with no serious practical applications. However, even with those extensions Datalog is far from becoming a language of choice. It is not well suited to formulate queries on attribute values, lacks aggregate functions and many other features of a fully fledged database query language.

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. Some authors (e.g. J.Ullman) consider this failure as an effect of bad terminology and advertising, but in our opinion the reasons are deeper. The obvious reason is that the Datalog community consists mainly of mathematicians who have never been engaged in big commercial software projects. Thus the main research pressure was on mathematical properties of Datalog programs rather than on attempts to show their usability in practical situations. As for many other research artifacts, no strong pressure on usability must result in poor usability. 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 applications 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 sylogisms and recursive rules.
- Limitations of 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. 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.

2.2.2 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 [SaPa97]. According to that paper, three different strategies of OO deductive query language design were visible:

- 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 [Li99,Li00]. OLOG is based on IQL – 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 being defined as collections of *objects* and relations as 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 query language, without an integrated programming language, methods are not supported.

DOQL [SP98] 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 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.

2.3 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). A typical example of a recursive procedure is a function calculating the factorial of a natural number. Recursive functions are also commonly used in processing of 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 database 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 recursion. Recursive procedures semantics requires introducing 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.

2.3.1 Recursive Procedures and Functions in Database Programming Languages

Database programming languages allow one to access data stored in a database directly, without unwieldy interfaces such as ODBC or embedded SQL, and process it on the database server. Unlike query languages, database programming languages are based on imperative programming paradigm (while query languages are declarative).

PL/SQL is one of the most powerful database programming languages, and the only one from commercial ones known to the authors, which does not limit the recursion depth. Other languages, such as Transact SQL and a language used for stored procedures in DB2, limit the recursion depth, typically to 16 nested calls. The limit on the recursion depth reduces the usability of a database programming language as a solution for recursive problems, because a solution that works for one set of data could fail for another one that would cause it to hit the recursion depth limit. PL/SQL syntax and semantics are described in detail in [Oracle].

Probably the most important application of recursive procedures in database programming languages is processing hierarchical structures. Thus, their applications are similar to transitive closure and fixed point equations. However, recursive functions are more flexible and expressive, as they can utilize sophisticated flow control constructs, as well as can change the database state. They can also encapsulate any complex code and can be reused in many places across database applications due to parameters. Recursive procedures suffer from some drawbacks, in particular, their execution can be slower than recursive queries because they are more difficult to optimize. Procedures require also some programming knowledge and experience from the users. Because procedures have to be stored in the database, besides some programming skill they require database administrator privileges.

Recursive procedures are used for tasks that are impossible to express directly as SQL queries. In PL/SQL there is common practice of using temporary tables to store calculation results, because in this language it is impossible to return a dynamic collection of complex objects. Although it is possible to implement dynamic collections, this would require separate implementation for each data type. This task is usually avoided by using various programming tricks, such as temporary tables mentioned before.

Example 9	**CREATE OR REPLACE PROCEDURE** find_depts (
	p_deptid **NUMBER**, p_tier **NUMBER DEFAULT** 1) **IS**
	  v_deptname **VARCHAR**(10);
	  CURSOR c1 **IS**
	    SELECT deptid, deptname **FROM** departments
	    WHERE superdept = p_deptid;
	  BEGIN
	    /* Get department name. */

```
  SELECT deptname INTO v_deptname FROM departments
  WHERE deptid = p_deptid;

  IF p_tier = 1 THEN
    dbms_output.put_line(v_deptname || ' is at top level');
  END IF;

  /* Find departments directly under this department. */
  FOR ee IN c1 LOOP
    dbms_output.put_line(v_deptname || ' includes ' ||
      ee.deptname || ' on tier ' || TO_CHAR(p_tier));

    /* Process next tiers in organization */
    find_depts(ee.deptid, p_tier + 1);  /* recursive call */
  END LOOP;
END;
```

A sample *find_depts* procedure, working with the data model presented in Fig.1, to a given department returns names of all sub-departments and the level of hierarchy they are on. The result is returned by dbms_output. The procedure first finds the id's and names of all departments subordinated to the given department. Then, it finds the name of the given department. If the department is on the top level it outputs the name together with string ' is at top level'. The next loop outputs the names of all the subordinate departments together with their hierarchy tiers, and invokes the procedure again, with new department id and the next tier in the hierarchy. After the execution of this procedure, the user obtains the output containing names of departments and their levels in the departments hierarchy.

Converting such a recursive procedure into a recursive function would require implementing some kind of collection as a function output. Linked list would be probably the easiest collection to implement, together with operations such as the sum of two lists.

Oracle "collections" are in fact static arrays, with no possibility of dynamically changing array size during runtime. Oracle offers some variations of those arrays, such as association arrays (that can be indexed by data of type other than integer), as well as operations that work on those "collections", such as addition. The arrays can hold fewer elements than their specified size but their size cannot be increased.

Oracle PL/SQL, despite being the most powerful and advanced database programming language, is significantly less powerful, than "conventional" programming languages. It is however well suited to tasks such as implementing triggers and complex data processing. It is not on par with modern programming languages when it concerns the ease of use of recursive processing.

2.3.2 Recursive Functions in XQuery

XQuery is a W3C standard of an XML query language. It supports recursive processing in the form of recursive functions. Unlike relational DBMSs, XQuery (at least its specification) does not impose any limits on recursion depth and mutually recursive functions.

The detailed specification of XQuery is available on the W3C website [W3Ca]. The language syntax and description is too complex to be included in this report. However, the language resembles some of the well-known programming languages, so the example below should be easy to follow.

The following sample function comes from the XML Query Use Cases document [W3Cb]. The function lists all the *part* elements connected with a *part* element received as the function parameter. The function processes an XML document with the structure described by the following DTD:

Example 10 – data structure

```
<!DOCTYPE partlist [
   <!ELEMENT partlist (part*)>
   <!ELEMENT part EMPTY>
   <!ATTLIST part
       partid  CDATA  #REQUIRED
       partof  CDATA  #IMPLIED
       name    CDATA  #REQUIRED>
]>
```

An element is a part of another element, if its' attribute *partof* is equal to another element's attribute *partid*. The function returning all *part* elements connected with a *part* element specified as its parameter is presented below:

Example 10 – function definition

```
declare function local:one_level($p as element()) as
element()
{
   <part partid="{ $p/@partid }"
       name="{ $p/@name }" >
      {
        for $s in doc("partlist.xml")//part
        where $s/@partof = $p/@partid
        return local:one_level($s)
      }
   </part>
};
```

The XQuery query below uses the above function to find the parts, which are not elements of other parts, and their subparts:

Example 10 – function call

```
<parttree>   {
   for $p in doc("partlist.xml")//part[empty(@partof)]
   return local:one_level($p)   }
</parttree>
```

Assume the following data set:

Example 10 – data set

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<partlist>
  <part partid="0" name="car"/>
  <part partid="1" partof="0" name="engine"/>
  <part partid="2" partof="0" name="door"/>
  <part partid="3" partof="1" name="piston"/>
  <part partid="4" partof="2" name="window"/>
  <part partid="5" partof="2" name="lock"/>
  <part partid="10" name="skateboard"/>
  <part partid="11" partof="10" name="board"/>
  <part partid="12" partof="10" name="wheel"/>
  <part partid="20" name="canoe"/>
</partlist>
```

For it the above query will return the following result set:

Example 10 – result set

```
<parttree>
   <part partid="0" name="car">
      <part partid="1" name="engine">
         <part partid="3" name="piston"/>
      </part>
      <part partid="2" name="door">
         <part partid="4" name="window"/>
         <part partid="5" name="lock"/>
      </part>
   </part>
   <part partid="10" name="skateboard">
      <part partid="11" name="board"/>
      <part partid="12" name="wheel"/>
   </part>
   <part partid="20" name="canoe"/>
</parttree>
```

The recursive processing capabilities introduced in XQuery are an important part of the language, although the authors of this report are not sure, whether the language design committee recognizes their importance. The fact that the XML Query Use Cases document contains only a single, short example of a recursive functions indicates otherwise.

In general, our critique of the XQuery solution concerns too exotic syntax with a lot of cryptic characters, tokens and constructs, which are illegible for a typical user. In this respect the solution is a step back in comparison to programming languages such as C++, Pascal or Java.

2.3.3 Recursive Data Processing Outside DBMS

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 ferature 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 infinite recursion), and thus it reduces or eliminates the need for drastic security measures (such as the recursion depth limitation mentioned earlier). 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.

Some database management systems, like Oracle, make it possible to create stored procedures in languages other than their native database programming languages. For example, Oracle allows a user to create stored procedures in Java. Those procedures still suffer from previously mentioned drawbacks. For example, Java stored procedures still have to use either JDBC or SQLJ to access data from the database.

Recursive queries have been the focus of study and research of many scientists, providing immense wealth of knowledge.

Fixpoint queries have been researched extensively. Among the most interesting issues is the expressive power of fixpoint queries. As shown in [Im86], one application of fixpoint operator suffices to express any query expressible (in relational calculus extended with fixpoint operator) with several alternations of fixpoint and negation. The author does not discuss the practical feasibility of such queries, which may be too difficult for users to write.

Another well-researched area is the extension of OO or relational databases with deductive capabilities. One of the examples of such research is the DOQL language, described in [SP98] (see 2.2.2), another – described in [DM92] (see 2.1.4) system extending SQL with recursive views based on fixpoint queries. The general problems – mainly the performance of DB access through various APIs - in coupling of relational databases with deductive engines are discussed in detail in [F98].

Evaluation models and optimization are also important fields. They have been covered in publications such as [HL95], discussing the evaluation of regular non-linear recursive queries, along with the methods of creating an effective query evaluation plans for different kinds of such queries. The problem of evaluation of recursive queries with non-stratified negation has been covered in [KSS95], again – with optimization methods. The authors of [LCSL98] propose recursive query optimization by transformation of logic programs, while the authors of [AVV97] discuss the connection between the computational complexity and fixpoint logic. Evaluation and optimization of general recursive queries by rewriting them into queries using materialized views has been described in [DG97] and [AGK99].

Transitive closure queries are also a well-researched problem. The expressiveness of transitive closure queries is discussed in [E91] – with a proof, that transitive closure combined with aggregate functions may be used to express some non-linear recursive queries. Incremental evaluation systems are very well researched (see 2.1.5), but transitive closure implementations for SQL utilizing semi-naive algorithms do exist, e.g. the system described in [N95]. Research into effective transitive closure query evaluation has also been conducted [QHK91], [ADJ90].

Nested relations are another related field of research. While flat relation is defined over a set of atomic attributes, a nested relation is defined over attributes which can include non-atomic ones. Such extensions to the relational algebra, along with the discussion of prototype implementations can be found in [He97] and [Ha97].

2.4 State of the Art Conclusions

Recursive query processing, despite the interest in the problem shown by both academic and commercial communities, is still in an immature stage. Most of the current research focuses on removing the inadequacies of existing solutions, but unfortunately, the chance of radically improving the situation is inexistent, as the inadequacies come from the foundations of each 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 materialized view maintenance).

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 w.r.t. asking 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 recursive query processing. 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. The rest of this report presents authors' proposed solution, based on the Stack Based Approach (SBA) to object-oriented query languages.

3 Stack-Based Approach – an Overview

3.1 Object Model

In SBA a query language is treated as a kind of a programming language. Thus evaluation of queries is based on mechanisms well known from programming languages. The approach precisely determines the semantics of query languages, their relationships with object-oriented concepts, constructs of imperative programming, and programming abstractions, including procedures, functional procedures, views, modules, etc.

SBA is defined for the general data store. In this report we explain our ideas for the simplest data store model (called M0 [Sub04]) where objects can contain other objects with no limitations on the level of the nesting of objects. There are also relationships between objects. The presented approach can be easily extended to comply with other notions like classes, inheritance, dynamic roles, etc.

SBA store models, in particular M0, are based on principles of object relativism and internal identification. Therefore, in the store each object has the following properties:

- Internal identifier (OID) that neither can be directly written in queries nor printed,
- External name (introduced by a programmer or the designer of the database) that is used to access the object from an application,
- Content that can be a value, a link, or a set of objects.

Let I be the set of internal identifiers, N be the set of external data names, and V be the set of atomic values, e.g. strings, pointers, blobs, etc. Atomic values include also codes of procedures, functions, methods, views and other procedural entities. Formally, objects in M0 are triples defined below ($i_1, i_2 \in I$, $n \in N$, and $v \in V$).

- Atomic objects have form $<i_1, n, v>$.
- Link objects have form $<i_1, n, i_2>$. An object is identified by i_1 and points at the object identified by i_2.
- Complex objects have form $<i_1, n, S>$, where S is a set of objects.

Note that this definition is recursive and it models nested objects with an arbitrary number of hierarchy levels.

In SBA an object store consists of:

- The structure of objects defined above.
- Internal identifiers of root objects (they are accessible from outside, i.e. they are starting points for querying).
- Constraints (e.g. the uniqueness of the internal identifiers, referential integrities, etc.).

Example database schema and the corresponding data store are depicted respectively in Fig.2 and Fig.3. The store contains the information on departments and their employees. Each employee has personal identification number, name, salary, and job. Each department has the attribute dName. The employees work in departments. Some employees may manage departments.

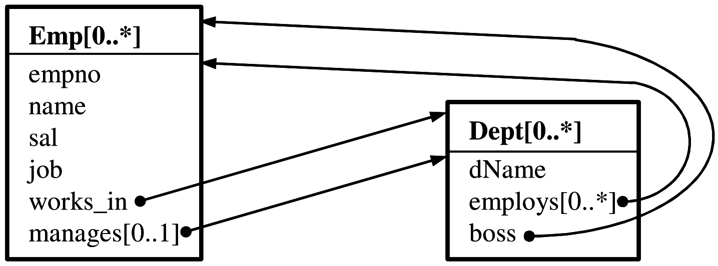

Fig. 2 Example database schema

Objects:

<i_1, *Emp*, {<i_2, *empno*, 123>, <i_3, *name*, "Smith">, <i_4, *sal*, 3500>, <i_5, *job*, "consultant">, <i_6, *works_in*, i_{20}>} >

<i_7, *Emp*, {<i_8, *empno*, 234>, <i_9, *name*, "White">, <i_{10}, *sal*, 2900>, <i_{11}, *job*, "developer">, <i_{12}, *works_in*, i_{20}> } >

<i_{13}, *Emp*, {<i_{14}, *empno*, 456>, <i_{15}, *name*, "Blake">, <i_{16}, *sal*, 3200>, <i_{17}, *job*, "manager">, <i_{18}, *works_in*, i_{20}>, <i_{19}, *manages*, i_{20}>} >

<i_{20}, *Dept*, {<i_{21}, *dName*, "IT">, <i_{22}, *employs*, i_1>, <i_{23}, *employs*, i_7>, <i_{24}, *employs*, i_{13}>, <i_{25}, *boss*, i_{13}>} >

ROOT = {i_1, i_7, i_{13}, i_{20}}

Fig. 3 Example of SBA data store

3.2 Environment Stack and Name Binding

The basis of SBA is the environment stack (ES). It is one of the most basic auxiliary data structures in programming languages. It supports the abstraction principle, which allows the programmer to consider the currently written piece of code to be independent of the context of its possible use. In SBA the environment stack consists of sections that contain sets of entities called binders. A binder is a construct that binds a name with a run-time object. Formally, it is a pair (n, i) (further written as n(i)) where n is an external name ($n \in N$) and i is the reference to an object ($i \in I$). In the following the binder concept will be extended to n(x), where x is any result returned by a query.

SBA respects the naming-scoping-binding principle, which means that each name occurring in a query is bound to a run-time entity (an object, an attribute, a method, a parameter, etc.) according to the scope of its name. The principle is supported by the environment stack. The process of name binding follows the "search for the top rule" and it returns the second component of a binder with the given name that is the closest to the top of ES and is visible within the scope of the name. Because names associated with binders are not unique, the binding can return multiple identifiers or values. In this way we deal with collections.

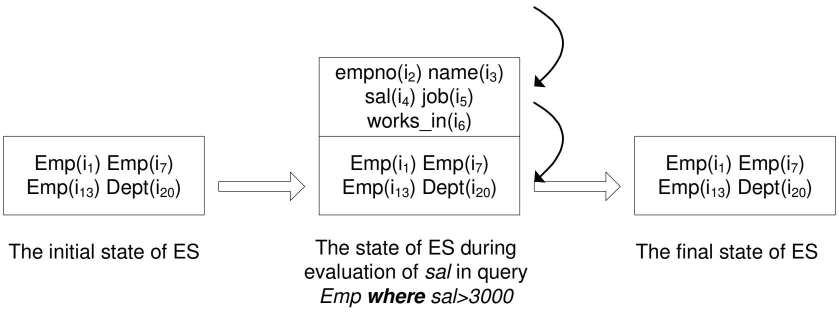

Fig. 4 Example of ES states during evaluation of a query

Some states of ES during evaluation of query *Emp* **where** *sal* > 3000 are depicted in Fig.4. Solid arrows indicate the order of name binding. At the beginning of the query evaluation ES consists only of the base section that contain binders to all root objects of the data store. There can be other base sections containing binders to the variables of the environment of the system, properties of the current session etc. If a query has no side effects, the state of the stack after the evaluation of the query is the same as before it. During query evaluation the stack grows and shrinks according to the nesting of the query.

New sections pushed onto ES are constructed by means of function *nested*. The function works in the following way:

- For the identifier of a link the function returns the set with the binder of the object the link points to.
- For a binder the function returns that the set with this binder.
- For a complex object the function returns the set of binders to the attributes of the object.
- For structures, *nested*(**struct**{x_1, x_2, ...}) = *nested*(x_1) ∪ *nested*(x_2) ∪ ...
- For other arguments the result of *nested* is empty.

Fig.4 contains the result of *nested* for i_1 (it returned a set: *empno*(i_2), *name*(i_3), *sal*(i_4), *job*(i_5), *works_in*(i_6)).

3.3 Stack Based Query Language (SBQL)

In this section we present the basis of the Stack Based Query Language (SBQL) [Sub85, Sub87, SBMS95, SKL95, Sub04]. The language has been first implemented in the LOQIS system [SMA90, Sub90, Sub91] then in several other systems. SBQL is based on the principle of compositionality, which means that more complex queries can be built of simpler ones. The description of the syntax of queries in SQBL follows.

- A name or a literal is a query; e.g. *name*, *Emp*, 2, "Black".
- $\sigma(q)$ is a query, where σ is a unary operator and q is a query; e.g. *sum*(*sal*), *sin*(*x*).
- $q_1 \tau q_2$ is a query, where q_1 and q_2 are queries and τ is a binary operator; e.g. 2+2, *Emp.sal, Emp* **where** (*sal* > 2000).

Therefore, in SBQL we can construct complex queries by composing them from simpler ones using unary and binary operators. With the exception of typing constrains the operators are orthogonal.

In SBA we divide operators into two main groups: algebraic and non-algebraic. In the following we describe the difference.

Algebraic operators. The operator is algebraic, if its evaluation does not use the environment stack. The evaluation of the query $q_1 \Delta q_2$ where Δ is algebraic operator looks as follows. Queries q_1 and q_2 are evaluated independently and the final result is a combination of these partial results depending on the semantics of operator Δ. Note that the key property of algebraic operators is that the order of evaluation of q_1 and q_2 does not matter.

Algebraic operators include string comparisons, Boolean and numerical operators, aggregate functions, operators on sets, bags and sequences (e.g. the union), comparisons, the Cartesian product, etc.

Non-algebraic operators. If the query $q_1 \theta q_2$ involves a non-algebraic operator θ, then q_2 is evaluated in the context determined by q_1. Thus, the order of evaluation of subqueries q_1 and q_2 is significant. The query $q_1 \theta q_2$ is evaluated as follows. A subquery q_2 is evaluated for each element r of the collection returned by q_1. Before each such evaluation ES is augmented with a new scope determined by *nested*(r). After the 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 r; the method of the combination depends on θ. Finally, these partial results are merged into the final result depending on the semantics of operator θ.

Non-algebraic operators include selection (operator **where**), projection/navigation (the dot), dependent join, quantifiers (**for all**, **for any**), various forms of transitive closures (**close by**, **leaves by**, **close unique by**, **leaves unique by**), etc.

Examples of queries in SQBL. Here we present examples of queries in SQBL addressing the example database shown in Fig.2.

- Get employees earning more then 30000:

  Emp **where** *sal* > 30000

- Get the name of the boss of IT department:

  (*Dept* **where** *dName* ="IT")**.***boss***.***Emp***.***name*

- Get the departments that have employees that have salary higher than the boss of the department:

  (*Dept* **as** *d*) **where** ((*d***.***employs***.***Emp* **as** *e*) ∃ (*e***.***salary* > d**.***boss***.***Emp***.***sal*))

3.4 Procedures, Functions, Methods and Views in SBQL

SBA supports ordinary procedures and functional procedures (functions). Procedures can be defined with or without parameters, can have side effects, local environment, and can be recursive. Procedures stored within classes and acting on objects' interiors are called methods. There are no restrictions on the computational complexity of procedures. The mechanism of the procedure call in SBA is the same as the mechanism of the procedure call in programming languages. When a procedure is called, the environment stack is augmented by a new section with the local environment of the procedure and its parameters. Next, the body of the procedure is executed. Then, if the procedure is functional its result is returned. Finally, the section with the environment of the procedure is removed from the top of the environment stack. Results of functional procedures belong to the same domain as results of queries and therefore, they can be called in queries.

Here is an example of a function in SBA (we omit typing).

```
proc wellPaidEmployeesOfDept(a) {
  create local avg((Dept where dName = a ).employs.Emp.sal) as avg_sal;
  return (Dept where dName = a).employs.Emp where salary > avg_sal;
}
```

This function returns the information on well-paid employees of the given department. The name of the department is the parameter of the function. The function has local variable *avg_sal* equal to the average salary in the indicated department. This function can be called in the following query:

wellPaidEmployeesOfDept("IT")**.**(*name*, *sal*)

This query returns names and salaries of all employees of IT department that earn more then the average in this department.

Concerning database views, in majority of approaches they are treated as first-class functional procedures. However, such an approach is inconsistent in many cases and may lead to warping user's intention. In SBA an alternative approach is proposed, in which the definer of a view can explicitly determine how updating operations on this view are to be performed. Such views can also be recursive and can have parameters. More details on SBQL views can be found in [KLPS02].

4 Transitive Closures in SBQL

A transitive closure operator in SBQL is one of the non-algebraic operators. It is defined as follows:

Syntax:

query ::= *query* **close by** *query*

Semantics:

The query q_1 **close by** q_2 is evaluated as follows:

- Query q_1 is evaluated first; the result should be of type *bag*. It is stored as the top element of the QRES stack (as usual). Let's call this element of the stack T.
- For each element $r \in T$ the following steps are carried out:
  - *nested(r)* is calculated, the result is put on top of ENVS; in M1-M3 models [Sub04], if r is a reference to an object, appropriate sections corresponding to classes and roles are put below this section as usual.
  - q_2 is calculated. It should return a *bag* of elements that are typologically compatible with elements returned by q_1. The result of q_2 is stored on top of QRES (as usual)
  - Union of T and the bag returned by q_2 is calculated; then elements returned by q_2 are added to the elements in T.
  - The top section of QRES is removed.
  - All sections put on ENVS in the first step are removed.

The elements returned by q_2 will be added to T, so they will be processed in some next step of processing elements $r \in T$. T will grow until the result of q_2 for the last element in T will be empty. At the end of this process, the top section of QRES will store the result of transitive closure.

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. In the Loqis system another operator – *distinct close by* – has been introduced to avoid infinite loops due to duplicates returned by q_2.

The transitive closure operator may be thought of as an equivalent of the fixpoint equation:

$$T = q_1 \cup T.\, q_2$$

The solution of this equation can be represented as an infinite union:

$$T = q_1 \cup q_1.q_2 \cup q_1.q_2.q_2 \cup q_1.q_2.q_2.q_2 \cup .....$$

As q_1 and q_2 can be any queries, simple or complex, the relation between elements that is used for transitive closure does not have to be explicitly stated in the database. It can be calculated on the fly during the query evaluation.

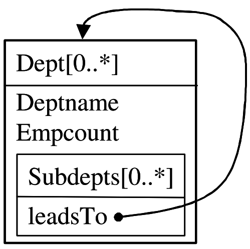

Fig. 5 Sample hierarchical data structure

A sample query finding all sub-departments of a department named “production” (using data structure presented in Fig.2) is shown below:

Example 11	(*Dept* **where** *Deptname*=”production”) **close by** (*Subdepts.leadsTo.Dept*)

Because relational databases are a particular case of the model M0, it is also possible to express the same query for a data model presented on Fig.1:

Example 12	(*departments* **where** *deptname*=”production”) **close by** ((*departments* **as** *x* **where** *x.superdept* = *deptid*).*x*)

Compared this with a similar Oracle query:

Example 13	**SELECT** deptid **FROM** departments **CONNECT BY** superdept = **PRIOR** deptid **START WITH** deptname = 'Production';

We argue that the SBQL query has an advantage: a subquery responsible for selecting the objects used as starting point in calculating the transitive closure, and a subquery specifying the relation are separate, not mixed with each other. This makes easier to compose and understand queries with transitive closures. When compared with DB2 query fulfilling the same function, the advantage of SBQL become even more obvious:

Example 14	**WITH** temptab(deptid, superdept) **AS** (
	SELECT root.deptid, root.superdept
	FROM departments root
	WHERE deptname='Production'
	UNION ALL
	    SELECT sub.deptid, sub.superdept
	    FROM departments sub, temptab super
	    WHERE sub.superdept = super.deptid
	) **SELECT** deptid **FROM** temptab

The DB2 query is much longer than SBQL and Oracle queries and requires declaration of a temporary table to store the data used in query evaluation.

SBQL queries utilizing a transitive closure have another advantage over similar queries in other DBMS: they can be used as building blocks for larger queries, including those performing update or delete operations. It is impossible with SQL queries.

Thanks to the possibility of using named values in queries, SBQL lets the user to perform any calculations as a part of processing the query. The calculations may range from simple (like calculating the level of hierarchy on which a particular object occurs) to complex (like calculating the square root of a given number). This greatly enhances the expressiveness of SBQL queries with transitive closure, allowing the user to formulate queries, which are impossible to express in Oracle version of SQL and would be very difficult (if possible at all) to express in DB2.

To show the use of named values defined within queries we introduce another data structure, Fig.3. It is a description of parts, similar to descriptions used in Bill of Material (BOM) applications.

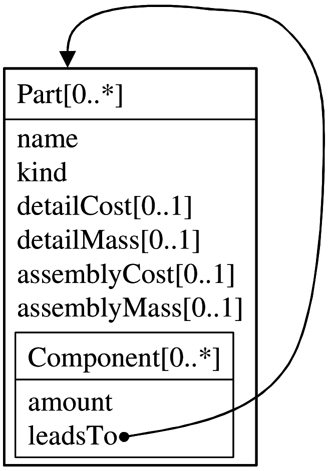

Fig. 6 Sample BOM data structure

The semantic meaning of the schema presented in Fig.3 is as follows. Each part has *name* and *kind*. If *kind* is "detail", the part has also *detailCost* and *detailMass* (the cost and mass of this part) and has no *assemblyCost*, *assemblyMass* attributes. If *kind* is "aggregate", the part has no *detailCost* and *detailMass*, but it has *assemblyCost* and *assemblyMass*. The attributes represent the cost of assembling this part and mass added to the mass of the components as the result of the assembly process). Aggregates have one or more Component sub-objects. Each Component has *amount* attribute (number of components of specific type in a part), and a pointer object *leadsTo*, showing the part used to construct this part.

One of the basic BOM problems – finding all components of a specific part, along with their amount required to make this part – may be formulated using the transitive closure:

Example 15	((*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 – which is also named *howMany* and paired with the “child” object.

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

Example 16	((((*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 suming 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 as the result of this query). There are many ways to accomplish this goal, among them:

- *leaves by* operator could be used in place of the *close by* operator. *leaves by* is a variant of the transitive closure operator, which returns only the “leaf” objects, that is, objects, which do not result in adding any further objects to the result set;
- Filtering out the non-leaf objects, by using the *kind* attribute:

Example 17	((((*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* **where** *kind*=”detail”) **as** *y*). (*y*, *sum*((*allEngineParts* **where** *x* = *y*).*howMany*)))

- Returning objects which do not have components:

Example 18	((((*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* **where not exists** *Component*) **as** *y*). (*y*, *sum*((*allEngineParts* **where** *x* = *y*).*howMany*)))

This task is one of the most typical BOM problems. It would be extremely difficult to formulate in any variant of SQL as a single query. The complexity of this task in SBQL – although still high – is possible to cope with, using grouping to divide the problem into smaller easier tasks.

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 [AB87] as an example of inadequacy and lack of flexibility of currently used query languages. In this case SBQL proves to be enough powerful.

Example 19	((((*Part* **where** *name*=”engine”) **as** *x*, (1 **as** *howMany*)) **close by** (*Component*.((*leadsTo.Part*) **as** *x*, (*amount***howMany*) **as** *howMany*))) **group as** *allEngineParts*). (*allEngineParts*. (**if** *x.kind*=”detail” **then** ((*howMany* * *x.detailCost*) **as** *c*, (*howMany* * *x.detailMass*) **as** *m*) **else** ((*howMany* * *x.assemblyCost*) **as** *c*, (*howMany** *x.assemblyMass*) **as** *m*))) **group as** *CostMassIncrease*). (**sum**(*CostMassIncrease.c*) **as** *engineCost*, **sum**(*CostMassIncrease.m*) **as** *engineMass*)

The transitive closure operator in SBQL allows the programmer to perform recursive calculations without extracting any information from the database – something that is impossible in other query languages. The example below calculates the square root of *a*, using the fixpoint equation $x = (a/x + x)/2$.

Example 20	((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)

The biggest problem with SBQL queries containing transitive closure (and many other complex queries as well) can be illustrated with the above queries: when performing complex tasks, queries become very complex. This may be the source of many problems with system maintenance, as queries become difficult to understand or modify. They may also become too complex for average application programmer to create, as they require good understanding of the semantics of SBQL.

Some of those problems can be avoided by using one of three slightly modified versions of the close by operator. The first modified operator is the *leaves by* operator. It returns only references to the leaf objects, instead of all objects in the hierarchy. Query from example 16 can be modified easily to return only the leaf (detail) parts, instead of all parts:

Example 21	((((*Part* **where** *name*=”engine”) **as** *x*, (1 **as** *howMany*)) **leaves 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*)))

Similarly to the query from example 17, this query returns all the “leaf” parts needed to create a part named “engine”. Instead of using *where* predicate to filter out the non-leaf objects, it uses the *leaves by* predicate to return only the leaf objects. There is a practical difference in those two approaches, which can be easily utilized in order to check database integrity (e.g. to find “leaf” objects which are not “detail” parts).

Cycles in the queried graph can be easily dealt with thanks to another modification of the *close by* operator – *close unique by*. This modification in each iteration removes duplicate object references as one of the evaluation steps. Removing duplicate object references breaks the cycles.

Let us consider a stock exchange database, storing information about share ownership, with data model shown on Figure 6 below:

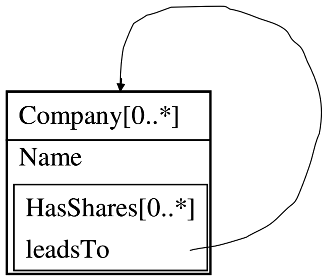

Fig. 7 Sample data structure for Example 18.

A company, described by a name may own shares in other companies (as we are not concerned with amount or value of those shares, they are omitted from the data structure). Those other companies may own shares in further companies (and so on). An important information to remember, is that it is possible for a company (lets call it “Company A”) to own shares in another company (“Company B”), which in turn owns shares in “Company A” (directly or by owning shares in yet another company). This creates a cycle in the graph. To find the names of all companies owned (directly or by proxy) by “Company A” we cannot use a simple query using *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 Example 22 instead:

Example 22	(*Company* **where** *Name*=”Company A” **close unique by** (*HasShares.leadsTo.Company*)).*Name*

The final modification of the *close by* operator is the *leaves unique by* operator. It is a combination of the two previous modification – it returns only leaf objects, while preventing problems with graph cycles.

Summing up, transitive closure operators in SBQL seem to be useful for relatively simple problems requiring recursion. For more difficult problems the use of transitive closure is theoretically possible, but not advised. A solution through transitive closures suffers from lack of facilities that would allow the user to divide the query into manageable parts corresponding to conceptual decomposition of the problem. It is not excluded, however, that this first impression concerning transitive closures can be relaxed after some training and experience.

5 Fixpoint Systems in SBQL

SBQL is very flexible and has the potential to support many different programming paradigms. Among others, SBQL implementations can provide querying capabilities similar to those presented by 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. There is a possibility to use a system of such equations with an arbitrary number of variables. One may argue that such fixpoint systems have limitations and differences when compared to Datalog, in particular the following:

- Datalog is used to deduce *facts*, using other *facts* and *rules* as a basis for such deduction. SBQL fixpoint systems are used to find objects or (complex) values that satisfy some conditions;
- Datalog is based on logic, thus in some of the evaluation paradigms it is possible to *prove* the deduction – present a proof, using defined facts and rules, that the derived facts are true. SBQL theoretical foundations lie elsewhere, and the possibility of proving the "deduction" was not among the chief concerns of the SBQL design;
- The equations in SBQL fixpoint systems (which can be thought of as equivalent to Datalog rules) may use any SBQL operators – in fact, the body of a fixpoint equation may be any valid SBQL query;
- SBQL does not put any constraints on the negation operation and does not assume CWA. However, negation is not the only operation, which may result in a query, which evaluation will result in an infinite loop. In SBQL it is assumed that the programmer is aware of the danger that goes with using such operators and takes appropriate care when using them.

These differences, however, 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. Perhaps deep comparison of both paradigms may display some advantages of Datalog, but currently they are not evident. Even if such advantages exist, the SBQL is not a closed artifact and can be improved to capture them. For these reasons we consider SBQL fixed point systems as a direct counterpart of Datalog programs.

The syntax of a SBQL fixpoint system is as follows:

fixpoint($x_{11}, x_{21},\ldots, x_{n1}$) { x_1 :- q_1; x_2 :- q_2; … x_m :- q_m; }

where:

- x_1 to x_m are names denoting variables in this equation system,
- x_{11} to x_{n1} are returned variables, $\{x_{11}, x_{21},\ldots, x_{n1}\} \subseteq \{x_1, x_2,\ldots, x_m\}$, i.e. not all variables from the equation system have to be returned,
- q_1 to q_m are SBQL queries dependent on variables $x_1, x_2,\ldots, x_m$;

The semantics of this language construct is the following:

1. Variables x_1 to x_m are initialized to empty *bags*;
2. Queries q_1 to q_m are evaluated;
3. Results of queries q_1 to q_m are assigned to variables x_1 to x_m, correspondingly;
4. Steps 2 and 3 are repeated, until results of queries q_1 to q_m are unchanged when compared to the same results from previous iteration; i.e. the fixed point is reached;
5. Variables $x_{11},x_{21},\ldots,x_{n1}$ are put on top of QRES as the structure of binders **struct**{ $x_{11}(v_{11})$}, $x_{21}(v_{21})$},…, $x_{n1}(v_{n1})$}, where $v_{11}, v_{21},\ldots, v_{n1}$ are calculated fixpoints of variables $x_{11}, x_{21},\ldots, x_{n1}$.

As queries q_1 to q_m can use variables x_1 to x_m in their bodies, the fixpoint system provides recursive capabilities: the result of an iteration depends on the result of the previous iteration.

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 uses data model shown in Fig.3):

Example 23	**fixpoint** (*parts*){ *parts* **:-** (*Part* **where** *name*=”engine”) **union** (*parts.Component.leadsTo.Part*);}

The query uses the single variable named *parts*. The variable is initialized as an empty *bag*. In the first step, the result of the following query:

(*Part* **where** *name*=”engine”) **union** (*parts.Component.leadsTo.Part*)

is assigned as value of this variable. As the variable *parts* is an empty bag, the assigned bag contains only *Part* objects, which are named “engine”. If there are no such objects, the evaluation stops (all variables in the fixpoint system are unchanged when compared to their values in previous iteration). Otherwise, each of the subsequent iterations will add new elements to the *parts* variable, as the result of evaluation of the path expression:

parts.Component.leadsTo.Part

which finds all *Components* of the objects currently stored in the *parts* variable. Once the point is reached, where no more components are added (as all the parts added in previous steps are "detail" parts), the evaluation will stop.

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 all unique engine elements:

Example 24	**distinct**(**fixpoint** (*parts*){ *parts* **:-** (*Part* **where** *name*="engine") **union** (*parts.Component.leadsTo.Part*);})

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":

Example 25	**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 to perform calculations, but never returned, 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 have to be calculated recursively – their purpose is to make the query easier to read and create.

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

Example 26	**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*); *detailsMass* **:-** *sum*((*engineParts* **where** *x.kind* = ”detail”). (*howMany*x.detailMass*)); *detailsCost* **:-** *sum*((*engineParts* **where** *x.kind* = ”detail”). (*howMany*x.detailCost*)); *addedMass* **:-** *sum*((*engineParts* **where** *x.kind* = ”aggregate”). (*howMany*x.assemblyMass*)); *addedCost***:-** *sum*((*engineParts* **where** *x.kind* = ”aggregate”). (*howMany*x.assemblyCost*)); *cost* **:-** *detailsCost* + *addedCost*; *mass* **:-** *detailsMass* + *addedMass*;}

Queries utilizing fixpoint queries, unlike those utilizing transitive closure, are capable of evaluating more than one recursive problem in each recursion step, in a manner similar to the Datalog. It is believed that such capabilities will allow users to formulate complex “business rules”. This topic may be an interesting area for further research, although most of the practical recursive problems authors are aware of can be solved using only a single recursion.

Similarly to transitive closure operator, fixpoint systems may be used to perform recursive calculations without referring to the database at all. The example below shows a fixpoint system version of Example 20 (calculating the square root of *a*):

Example 27	**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 present in Datalog, such as the difficulty with formulating queries based on complex objects. When compared with queries utilizing transitive closure operator, especially more complex ones, fixpoint systems seem to be more readable (as decomposition of the problem is easier). This feature may be more important to the users, than their theoretically greater expressiveness.

6 Recursive Procedures and Views in SBQL

SBQL philosophy allows for seamless integration of imperative language constructs, including recursive procedures and functions with query operators. This allows users to easily utilize the most popular recursive processing technique, without sacrificing any of the benefits of query language. In contrast to popular programming languages the new quality that SBQL introduces 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.

The only difference between procedures and functional procedures is that functional procedures contain a *return* statement. There is no separate syntax for them. The syntax for SBQL procedures is shown below.

Syntax:

proc	::=	**procedure** *procName* { *statements* }
proc	::=	**procedure** *procName* ( ) { *statements* }
proc	::=	**procedure** *procName* (*formalParams*) { *statements* }
procName	::=	*name*
formalParams	::=	*formalParam* | *formalParam*; *formalParams*
formalParam	::=	*name* | **in** *name* | **out** *name*
statement	::=	*imperativeOperator query*; | **return [***query*]
procCall	::=	*procName* | *procName* ( ) | *procName* (*actParams*)
actParams	::=	*actParam* | *actParam*; *actParams*
actParam	::=	*query*

Semantics:

When a procedure is called, the statements in the procedure are evaluated in the consecutive order. When the *return* statement is reached, the result of the query being parameter of that statement is put on top of QRES. If no *return* statement is reached before the end of procedure evaluation, the procedure returns nothing (thus it is not a functional procedure).

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, see [Subieta04]. SBQL includes many such imperative operators (object creation, flow control statements, loops, etc.).

The simplest recursive procedure consists of a single return statement, which depends on a procedure parameter either returning an empty collection or returning the result from invocation of the same procedure with different parameter value. A sample recursive procedure finding all components of a specific part, along with their amount required to make this part, is shown below (see Fig.3):

Example 28	**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 (named *myPartsHowMany*). Each structure contains a reference to a part object named *c* and the amount of parts of this type named *howMany*. The example below shows that this procedure may take a collection of parameters, instead of a single parameter, without any alterations.

Example 29	*SubPartsHowMany* (**bag**((*Part* **where** *name* = ”engine”) **as** *c*, 68 **as** *howMany*), (*Part* **where** *name* = ”gear box”) **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), and calculations within a single procedure can be performed in easy to comprehend steps, comparable to the simplicity of fixpoint systems. A procedure calculating the cost and mass of a part illustrating this possibility is shown in Example 30. The procedure utilizes the previously defined *SubPartsHowMany* procedure in order to perform the recursive processing (which is identical for both tasks) and then performs calculations, utilizing local variables (created with *create local* statements).

Example 30	**procedure** *CostAndMass*(*myPartsHowMany*) { **if not exists**(*myPartsHowMany*) **then return bag**(); **create local** *SubPartsHowMany*(*myPartsHowMany*) **as** *parts*; **create local sum**((*parts* **where** *c.kind*=”detail”). (*howMany**c.detailMass*)) **as** *detailsMass*; **create local sum**((*parts* **where** *c.kind*=”detail”). (*howMany**c.detailCost*)) **as** *detailsCost*; **create local sum**((*parts* **where** *c.kind*=”aggregate”). (*howMany**c.assemblyMass*)) **as** *addedMass*; **create local sum**((*parts* **where** *c.kind*=”aggregate”). (*howMany**c.assemblyCost*)) **as** *addedCost*; **return struct**((*addedCost*+*detailsCost*) **as** *cost*, (*addedMass*+*detailsMass*) **as** *mass*))}

Recursive procedures in SBQL offer many advantages, when compared to stored procedures in relational DBMSs. Most of them are consequences of the fact that recursive procedures in SBQL are a natural extension of the SBA, working on the same principles and evaluated by the same evaluation engine – unlike relational systems, where stored procedures are an addition to the system, but are evaluated separately from SQL queries. The main advantages of SBQL recursive procedures are the following:

- SBQL queries are valid as expressions, procedure parameters, etc.;
- The type system (if used) is the same;
- There is no impedance mismatch;

SBQL procedures may be also used as a way to reuse queries or fixpoint systems. Instead of writing a stand-alone fixpoint system for a single use, it is possible to write a procedure utilizing the fixpoint system, while providing a way to parameterize it easily. A sample procedure doing this is shown below:

Example 31	**procedure** *subParts*(*whatPart*) {
	return distinct(**fixpoint** (*parts*){
	parts **:-** *whatPart* **union**
	(*parts.Component.leadsTo.Part*); })}

The procedure takes a single parameter *whatPart*, which is used in the fixpoint system as a starting point for calculations. The fixpoint system is a part of query in the return statement. In this way the fixpoint system can be invoked with different starting points, with no need for rewriting it each time.

SBQL views are based on procedures, and as such can be recursive or utilize recursive procedures, transitive closure or fixpoint systems to provide user with the desired data. A sample read-only view, returning subparts of a component named "engine" along with the number of those subparts required to make the engine is shown below. The view utilizes fixpoint in order to "flatten" the hierarchy.

Example 32	**create view** *EnginePartsDef* {
	virtual objects *EngineParts* {
	return distinct(**fixpoint** (*parts*) {
	parts **:-** (*Part* **where** *name*="engine")
	union (*parts.Component.leadsTo.Part*);
	}) **as** *b*;
	}
	on retrieve do return b; }

SBQL views are discussed in detail in [KLPS02], [KLS03a], [KLS03b] and [KS04], as well as in [SP01] and [Sub04].

7 Optimization Opportunities

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. Clearly defined semantics of SBQL allows for a systematic and disciplined approach to this problem. The optimization possibilities for all three approaches are described below.

7.1 Transitive Closure

The basic technique of SBQL query optimization, including transitive closure queries is query rewriting. This technique is based on the idea of creating an equivalent query (one, that will produce the same final result) that will be evaluated faster. This can be achieved by starting with the original query's syntactic tree and modifying it in order to avoid multiple evaluations of the same independent subquery. It is a generalization of a well known and widely used in relational DBMSs method of factoring selection before join operation. This technique has been presented in detail in [PS99], [PK00], [Plo00], [PS01a] and [PS01b]].

Utilizing indices is another technique, which may be useful for some of the queries utilizing transitive closure. Indices are introduced into queries during query rewriting process, when subqueries, which can be replaced by index calls, are identified. As for now, no in-depth research on index-based query optimization in SBQL has been conducted.

7.2 Fixpoint Systems

Evaluation of a fixpoint system may be an expensive process. It may be optimized in two ways. First, each individual query can be a subject to the query rewriting process. This may reduce the evaluation cost of the query, and as a consequence, of the entire fixpoint set.

Second optimization technique, which may be utilized, is an adaptation of the improved seminaive technique known from Datalog. The main idea of this technique is to divide the fixpoint equations (or rules, in the case of Datalog) into groups, which may be evaluated independently from the other groups, utilizing only the final results of previous group evaluation. The concept is shown on Fig.4 below:

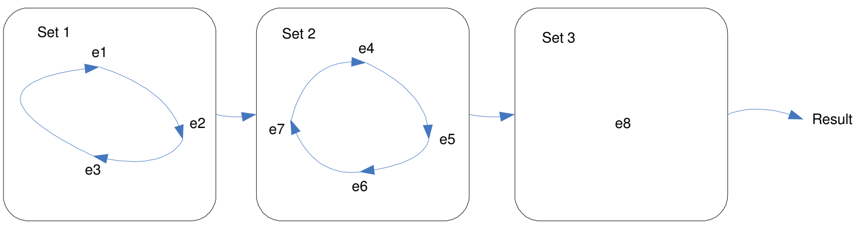

Fig. 8 The seminaive evaluation optimization process concept

A fixpoint set of 8 equations (*e1..e8*) is divided into three sets of equations – Set 1 (*e1,e2,e3*), Set 2 (*e4,e5,e6,e7*) and Set 3(*e8*). Set 1 is evaluated recursively first, as equations from this set are referencing only to variables calculated in equations belonging to this set (making this set independent from other equations). When the equations in Set 1 reach the fixpoint, evaluation of Set 2 begins, as equations from Set 2 are referencing only to variables defined in Set 2 and one (ideally) or more variables defined in Set 1. When the fixpoint is reached in Set 2, evaluation of Set 3 is performed, until the result is found.

This technique allows avoiding unnecessary calculations during the fixpoint evaluation. As not all equations need to be evaluated in each step, the cost of the evaluation may be significantly reduced. This is not only important for complex fixpoint systems, with multiple concurrently evaluated equations, but also for fixpoint systems similar to those presented in this report – with equations serving as a tool for a simple recursive problem decomposition.

The following example shows a situation, in which we optimize a fixpoint set with simple recursive problem decomposition. There is a single equation which needs to be calculated recursively and seven equations, which need to be calculated only once (the fixpoint used comes from Example 24):

Example 33	**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*); *detailsMass* **:-** *sum*((*engineParts* **where** *x.kind*="detail"). (*howMany**x.detailMass*)); *detailsCost* **:-** *sum*((*engineParts* **where** *x.kind*="detail"). (*howMany**x.detailCost*)); *addedMass* **:-** *sum*((*engineParts* **where** *x.kind*="aggregate"). (*howMany**x.assemblyMass*)); *addedCost***:-** *sum*((*engineParts* **where** *x.kind*="aggregate"). (*howMany**x.assemblyCost*)); *cost* **:-** *detailsCost* + *addedCost*; *mass* **:-** *detailsMass* + *addedMass*;}

Without optimization, each equation would have to be calculated in each recursion step. This means, that if the fixpoint was reached in 10 steps, the "non recursive" equations would be evaluated 10 times each – it means 63 unnecessary equation evaluations.

Now, lets consider optimization of the fixpoint:

- The equation *engine* needs to be evaluated first. It does not reference to variables defined by other equations – so it can be evaluated as a group on its own. It does not reference itself, so the fixpoint will be reached in the first evaluation step;
- The *engineParts* equation references the *engine* variable (defined by *engine* equation) and itself. As the *engine* variable was calculated in the previous group, the *engineParts* equation will be evaluated next ;

- The *detailsMass, detailsCost, addedMass and addedCost* variables are defined by equations, which reference only the *engineParts* variable. Each can be evaluated as a separate group. Order in which they will be evaluated is not important – their order in the fixpoint set may be used;
- The *cost* variable is defined by equation, which references *detailsCost* and *addedCost* variables. It may be evaluated no sooner than equations defining those two variables, as a separate group.
- The *mass* variable is defined by equation, which references *detailsMass* and *addedMass* variables. It may be evaluated no sooner than equations defining those two variables, as a separate groups.

This time only the group containing the *engineParts* equation would be evaluated several times. Other groups would reach the fixpoints of their equations in single evaluation step each, thus saving evaluation time.

Probably the magic sets technique, also known from Datalog ([Ull90]) can be adapted to SBQL queries – further research in this direction is needed.

7.3 Recursive Procedures

Recursive procedures are very different from normal queries and fixpoint systems when it comes to the optimization problem. Unlike the other approaches, recursive procedures are based on the imperative, non-declarative paradigm. While it is possible to optimize fragments of recursive procedures, by query rewriting, or by using techniques typical for typical programming languages, it is impossible to optimize a problem solution as a whole. The solution of a recursive problem may be spread over several recursive procedures, which in turn may be parts of solutions of several different problems. What is even more important, techniques used in imperative programs optimization are not very effective, when compared to optimization algorithms used in data bases. In particular, removing dead code is not an optimization, it is rather rectifying programmer's mistakes, and the results achieved with techniques like loop unrolling are negligent when compared to query evaluation cost.

Caching results of procedures is another technique, which might be useful, but only in certain situations. If an expensive (time-wise) procedure is often called with the same parameter values, and data the procedure is using in the evaluation process does not change frequently, the procedure results could be cached and used as needed, instead of calling the procedure. In the authors' opinion, however, this technique should be used with care, as the cost of identifying such cases may be prohibitive. Probably the task of identifying those cases could be assigned to application developer or database administrator.

8 Conclusions

In this report we have presented and compared several implemented and postulated techniques for expressing recursive problems in database query and programming languages.

In Chapter 2 we have presented the current State of the Art. Despite the effort put into research on recursive queries, its results are not widely used outside academia. Commercial DBMSs, which implement some sort of recursive queries offer either a very limited form of recursion, or do not offer any means of making their r recursive querying facilities easy (e.g. through problem decomposition). Academia research produces either systems with very poor performance (due to the problems connected with extending existing DBMSs with new features, or creating new ones), or systems difficult to use by industry programmers due to the theoretical knowledge required from their users and difficulty in formulating queries typical for day-to-day work (this mainly applies to the various Datalog variants).

In Chapter 3 we have presented the foundation of our proposed solution – the SBQL query language. SBQL semantics are not based on any kind of relational or object algebra – instead they are derived from the principles of imperative programming languages. This language is able to handle macroscopic statements – a requirement for a query language. It also has clear semantics, allowing for easy query manipulation and optimization, using techniques both from query languages and imperative programming languages.

In Chapter 4 we have presented the transitive closure operator in SBQL - its semantics in the Stack Based Approach and pragmatics – using examples known from literature. Similarly, we have presented the fixpoint equation systems in SBQL in Chapter 5. Both approaches integrate seamlessly with the query language and are easy to use.

Chapter 6 contains the presentation of recursive functions and views in SBQL. Recursive functions (and views based on those functions) are an integral feature of SBQL and offer programmers the ease of use and functionality previously unknown in database programming languages – including truly seamless integration (like the possibility of accepting a query as the parameter for procedure).

Chapter 7 presents a short discussion for optimization possibilities. Recursive query optimization is a difficult problem, but a lot of already well-known and researched techniques may be used or adapted for SBQL.

SBQL provides the programmer with powerful recursion facilities, suitable for any task – from the simplest tasks, which are possible to solve using queries with transitive closure, to the most complicated tasks, which are possible to solve with fixpoint systems and recursive procedures. The construction of the language allows those facilities to be used to process any kind of data – from data stored in relational databases to XML files. This gives SBQL a clear advantage over other solutions presented in this report.

Bibliography

[AHV95] S.Abiteboul, R.Hull, V.Vianu. Foundations of Databases, ch. 12, Addison-Wesley (1995)

[AU79] A.V.Aho, J.D.Ullman: The Universality of Data Retrieval Languages, Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages: 110-120, ACM (1979)

[AB87] M.P.Atkinson, P.Buneman. Types and Persistence in Database Programming Languages. ACM Computing Surveys 19(2):105-190, ACM (1987)

[Ban78] F.Bancilhon: On the Completeness of Query Languages for Relational Data Bases. Lecture Notes in Computer Science 64: 112-123, Springer (1978)

[CGT89] 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)

[FJR92] 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)

[GMN84] H.Gallaire, J.Minker, J.-M.Nicolas: Logic and Databases: A Deductive Approach. ACM Comput. Surv. 16(2): 153-185, ACM (1984)

[KLPS02] H.Kozankiewicz, J.Leszczyłowski, J.Płodzień, K.Subieta. Updateable Object Views. Institute of Computer Science, Polish Academy of Sciences, Report 950, Warsaw, Poland (2002)

[KLS03a] H.Kozankiewicz, J.Leszczyłowski, K.Subieta. Updateable XML Views. Proc. of Advances in Databases and Information Systems (ADBIS), , Lecture Notes in Computer Science 2798: 385-399, Springer (2003)

[KLS03b] H.Kozankiewicz, J.Leszczyłowski, K.Subieta. Implementing Mediators through Virtual Updateable Views. Proc. of the 5th International Workshop on Engineering Federated Information Systems (EFIS): 52-62, IOS Press, Coventry, UK (2003)

[KS04] 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, to appear

[Oracle] PL/SQL User's Guide and Reference (9.2), Oracle, available online: http://www.oracle.com/pls/db92/db92.docindex

[PS99] J.Płodzień, K.Subieta. Optimization of Object-Oriented Queries by Factoring Out Independent Subqueries. Institute of Computer Science Polish Academy of Sciences, Report 889, November 1999

[PK00] J.Płodzień, A.Kraken. Object Query Optimization through Detecting Independent Subqueries. Information Systems, Pergamon Press, September 2000.

[Plo00] J.Płodzień. Optimization Methods in Object Query Languages. Ph.D. Thesis. Institute of Computer Science, Polish Academy of Sciences (2000), available via http://www.ipipan.waw.pl/~jpl

[PS01a] 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)

[PS01b] J.Płodzień, K.Subieta. Static Analysis of Queries as a Tool for Static Optimization, Proceedings of IDEAS Conference: 117-122, IEEE Computer Society (2001)

[Sub85] K.Subieta . Semantics of Query Languages for Network Databases, ACM Transactions on Database Systems, Vol. 10, No. 3: 347-394, ACM (1985)

[Sub87] K.Subieta. Denotational Semantics of Query Languages, Information Systems, Vol. 12, No. 1 (1987)

[SMA90] K.Subieta, M.Missala, K.Anacki "The LOQIS System. Description and Programmer Manual", Institute of Computer Science, Polish Academy of Sciences, Report 695, Warsaw, November 1990

[Sub91] K.Subieta. LOQIS: The Object-Oriented Database Programming System, Proceedings of the 1st Intl. East/West Database Workshop on Next Generation Information System Technology, Lecture Notes in Computer Science 504: 403-421, Springer (1991)

[SP01] K.Subieta, J.Płodzień Object Views and Query Modification, Databases and Information Systems: 3-14, Kluwer Academic Publishers, (2001)

[SBMS95] 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)

[SKL95] K.Subieta, Y.Kambayashi, and J.Leszczyłowski. Procedures in Object-Oriented Query Languages. Proc. VLDB Conf., 182-193, Morgan Kaufmann (1995)

[Sub04] K.Subieta. Theory and Construction of Object-Oriented Query

Languages. Editors of Polish-Japanese Institute of Information Technology, Warsaw 2004, 522 pages, (in Polish)

[TH91] 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)

[Ull90] J.D.Ullman. Principles of Database and Knowledge-Base Systems, volume II, ch. 13, W H Freeman, 1990

[UW97] J.D.Ullman, J.Widom. A First Course in Database Systems, ch.5, Prentice Hall, 1997

[W3Ca] XQuery 1.0: An XML Query Language, http://www.w3.org/TR/xquery/, working draft

[W3Cb] XML Query Use Cases, http://www.w3.org/TR/xquery-use-cases/, working draft

[YM89] 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)

[SaPa97] 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)

[Li93] Z. Li, K. A. Ross, On the Cost of Transitive Closures in Relational Databases, Columbia University Technical Report CUCS-004-93, April, 1993

[Do97] G. Dong, L. Libkin, L. Wong, Properties of Languages That Make Recursive Views Unmaintainable, available online: http://citeseer.ist.psu.edu/4739.html

[Do99] G. Dong, L. Libkin, J. Su, L. Wong, Maintaining transitive closure of graphs in SQL, Int. Journal of Information Technology, Singapore Computer Society (1999)

[Li97] L. Libkin and L. Wong, Incremental recomputation of recursive queries with nested sets and aggregate functions. LNCS 1369: Proceedings of 6th International Workshop on Database Programming Languages: 222-238 Springer Verlag (1997)

[Im86] N. Immerman, Relational Queries Computable in Polynominal Time, ACM Symposium on Theory of Computing: 147-152, ACM (1982)

[SP98] 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)

[DM92] M. Dobrovnik, R. T. Mittermeir, Architectural Considerations for Extending a Relational DBMS with Deductive Capabilities, Proceedings of SPSE '92, Springer Verlag (1992)

[HL95] J. Han, L. V. S. Lakshmanan, Evaluation of Regular Nonlinear Recursion by Deductive Database Techniques, Information Systems 20, Pergamon Press (1995)

[KSS95] D. B. Kemp, D. Srivastava, P. J. Stuckey, Bottom-up Evaluation and Query Optimization of Well-Founded Models, Theoretical Computer Science 146: 145-184, Elsevier Science Publishers. (1995).

[LCSL98] J. Lu, L. Chen, K. Symara, J. Lu, Recursive Query Rewriting by Transforming Logic Program, International Workshops on Logic-based Program Synthesis and Transformation, Manchester, United Kingdom, June 1998.

[AVV97] S. Abiteboul, M. Y. Vardi, V. Vianu, Fixpoint Logics, Relational Machines and Computational Complexity, Journal of ACM, ACM (1997)

[F98] J. Freire, Practical Problems in Coupling Deductive Engines with Relational Databases, Proceedings of the 5th KRDB Workshop, Seattle 1998, Swiss Life, Information Systems Research, Report 18, Zurich 1998

[DG97] O. M. Duschka, M. R. Genesereth, Answering Recursive Queries Using Views, Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Conference on Principles of Database Systems: 109-116, ACM Press (1997)

[AGK99] F. N. Afrati, M. Gergatsoulis, T. Kavalieros, Answering Queries Using Materialized Views with Disjunctions, Lecture Notes in Computer Science 1540: 435-452, Springer (1999)

[N95] B. Nag, Implementing Generalized Transitive Closure in the Paradise Geographical Information System, available online: http://citeseer.ist.psu.edu/nag95implementing.html

[He97] H. He, Implementation of Nested Relations in a Database Programming Language, available online: http://www.cs.mcgill.ca/~tim/cv/students.html

[Ha97] B. Hao, Implementation of The Nested Relational Algebra in Java, available online: http://www.cs.mcgill.ca/~tim/cv/students.html

[QHK91] G. Z. Qadah, L. J. Henschen, J. J. Kim, Efficient Algorithms for the

Instantiated Transitive Closure Queries, IEEE Transactions on Software Engineering archive, Volume 17 , Issue 3, 1991

[ADJ90] R. Agrawal, S. Dar, H. V. Jagadish: Direct Transitive Closure Algorithms: Design and Performance Evaluation, ACM Trans. Database Syst. 15(3): 427-458 (1990)

Pracę zgłosił: Mieczysław Kłopotek

Adres autorów:
Tomasz Pieciukiewicz [*]
Kazimierz Subieta [#*]
[#]) Institute of Computer Science, Polish Academy of Sciences
ul.Ordona 21, 01-237 Warszawa, Poland

[*]) Polish-Japanese Institute of Information Technology
ul.Koszykowa 86, 02-008 Warszawa, Poland

Symbol klasyfikacji rzeczowej: H.2.3, H.2.4, H.2.5.

Na prawach rękopisu
Printed as manuscript

Nakład 100 egzemplarzy. Papier kserograficzny klasy III. Oddano do druku w listopadzie 2004 r. Wydawnictwo IPI PAN. ISSN 0138-0648