



Polish-Japanese Institute
of Information Technology

Tomasz Pieciukiewicz

Recursive Queries in Databases

Ph.D. Thesis

Submitted to the Senate of the Polish-Japanese Institute of Information
Technology

Advisor:
Prof. Kazimierz Subieta

Warsaw, November 2009

Table of Contents

Abstract	8
Extended Abstract (in Polish)	9
I. Introduction.....	12
1. Motivation	12
2. Transitive Closures, Fixpoint equation systems and recursion.....	14
3. Key Work Objectives	14
4. Thesis Structure.....	15
II. Recursive Query Processing - the State-of-the-Art.....	16
1. Transitive Closure	16
i. Recursive Queries in Oracle.....	17
ii. Recursive Queries in SQL-99.....	21
iii. Recursive Queries in DB2	23
i. Recursive Queries in Microsoft SQL Server	26
ii. Other transitive closure implementations	27
iii. Caching Transitive Closure Query Results	28
2. Fixed Point Equations.....	29
i. Fixed Point Equations in Datalog.....	29
ii. Deductive Object-Oriented Databases.....	34
3. Recursive Procedures and Functions	36
i. Recursive Procedures and Functions in Database Programming Languages.....	36
ii. Recursive Functions in XQuery	39
iii. Recursive Data Processing Outside DBMS.....	42
4. State of the Art Conclusions.....	44
III. Stack-Based Approach - an Overview	46

1. Query Language for Recursive Processing	46
2. Object Model.....	46
3. Environment Stack and Name Binding.....	49
4. Stack Based Query Language (SBQL).....	50
i. Algebraic operators	51
ii. Non-algebraic operators.....	51
iii. Examples of queries in SBQL.....	51
iv. Procedures, Functions, Methods and Views in SBQL	52
IV. Transitive Closure Operator in SBQL.....	54
1. Transitive Closure Operators – syntax and semantics	54
i. The <i>close by</i> operator	54
ii. The <i>close unique by</i> operator	55
iii. The <i>leaves by</i> operator.....	57
iv. The <i>leaves unique by</i> operator	59
2. Transitive Closure – type checking.....	61
3. Transitive Closure – stop problem	62
V. Transitive Closure operator in SBQL – usage examples.....	64
1. Bill of Material	64
2. Workflow.....	68
VI. Optimization of Transitive Closure.....	73
1. Optimization by rewriting queries	73
i. Query optimization in SBA – general framework.....	73
ii. Factoring out independent subqueries	74
iii. Pushing out selections before the transitive closure operator	76
2. Other methods for Transitive Closure query optimization.....	78
i. Query result caching.....	78

ii.	Materialization of transitive closure	80
3.	Summary	80
VII.	Fixpoint Systems in SBQL	82
1.	Fixpoint Systems – Syntax and Semantics.....	82
i.	Syntax.....	82
ii.	Semantics.....	82
iii.	Fixpoint Systems and persistence.....	83
2.	Fixpoint Systems – type checking	83
iv.	Automated type inference.....	84
v.	Fixpoint systems type checking – recommended approach	87
VIII.	Fixpoint Systems in SBQL - usage examples.....	89
1.	Bill of Material	89
2.	Workflow.....	91
IX.	Optimization of Fixpoint Systems	95
1.	Fixpoint system-specific optimization method.....	95
i.	Stratified evaluation	95
ii.	Stratification algorithm.....	96
iii.	Example.....	97
2.	Standard methods used for query optimization.....	99
i.	Factoring out subqueries	99
ii.	Pushing out selection.....	101
3.	Synergy between stratification and factoring out independent subqueries.....	102
4.	Detection of non-recursive equations	104
X.	Recursive procedures and views in SBQL.....	107
1.	Procedures –Syntax and Semantics	107
2.	Views	108

XI.	Recursive functions and views in SBQL – usage examples	110
1.	Recursive functions	110
2.	Recursive Views.....	112
XII.	Optimization of recursive procedures.....	114
1.	Removal of shared sub-expressions.....	115
XIII.	Transitive Closures, Fixpoint Equation Systems and Recursive Functions – an Attempt at Comparison	117
1.	Expressive Power.....	117
2.	Optimization Potential	117
3.	Familiarity and Ease of Use	118
4.	Comparison Conclusions	119
XIV.	Implementation of Transitive Closure operators in ODRA DBMS	121
1.	Lexer and Parser	122
2.	Context analyzer.....	122
3.	Optimizer.....	122
4.	Code generator.....	123
5.	Interpreter.....	123
XV.	Implementation of Fixpoint Systems in ODRA DBMS.....	124
1.	Lexer and Parser	124
2.	Context analyzer.....	124
3.	Optimizer.....	125
4.	Code generator.....	125
5.	Interpreter.....	125
XVI.	Further research.....	126
1.	Datalog-like rules.....	126
2.	Recursive queries in distributed databases	126

3.	Alternative storage models	127
4.	Parametrized views	127
XVII.	Conclusions	128
XVIII.	List of Figures	129
XIX.	List of Examples	130
XX.	Bibliography.....	132

Abstract

Recursive processing is one of the most important programming and problem-solving paradigms. Many problems have solutions that can easily be expressed in terms of recursion. Database systems present the area in which recursive processing may be very useful, as information stored in them could often be interpreted as a graph of some sort that can be the subject of recursive processing. Many applications could require traversing of such graphs by means of queries. We can mention here applications like Bill of Material, processing of various kinds of networks (transportation, social), workflows, semi-structured data etc.

Support for recursive queries in databases is usually not satisfactory. The facilities for recursive processing are missing altogether or they are (often intentionally) limited. This especially concerns variants of SQL. On the other hand, query languages from the Datalog family provide extensive recursive querying capabilities, but are very limited concerning other features and not very popular among database developers. Actually, a fully fledged commercial database system based on Datalog does not exist.

This thesis presents three approaches to recursive query processing implemented for the Stack Based Query Language (SBQL), namely, transitive closures, fixpoint equation systems and recursive functions. Due to them SBQL offers very powerful and flexible recursive querying capabilities. The introduced recursive processing options and mechanisms are fully orthogonal to the other features of the language.

The thesis presents assumptions of each recursive processing approach and discusses the most important aspects of them. The syntax and semantics are presented for four different variants of transitive closures, as well as for fixpoint equation systems. Pragmatics of each of recursive processing paradigms are discussed, using examples to illustrate the relative merits of each approach. Finally – as recursive queries may be very expensive to evaluate - possible optimization techniques are discussed.

The thesis also discusses the implementation of transitive closures and fixpoint systems within the ODRA database management system.

Extended Abstract (in Polish)

Zapytania Rekurencyjne w Bazach Danych

Wiele zadań istotnych z punktu widzenia informatyki oraz biznesu to zadania rekurencyjne. Bazy danych są jednym z obszarów, w których możliwość przetwarzania rekurencyjnego może być bardzo przydatna – informacja przechowywana w bazach danych często może być interpretowana jako struktura drzewiasta albo inny rodzaj grafu, a potrzeby użytkowników mogą wymagać przejścia po danym grafie. Zastosowania takie jak Bill of Material, różnego rodzaju sieci (od społecznościowych, przez transportowe po telekomunikacyjne), grafy przepływu pracy, struktury organizacyjne, dane półstrukturalne (np. pliki XML) zaimportowane do baz danych itp. są naturalnymi obszarami zastosowań dla zapytań rekurencyjnych.

Rekurencyjne zapytania nie są stosowane tak często, jak wskazywałaby na to mnogość potencjalnych obszarów zastosowań. Standardy SQL-89 i SQL-92 nie przewidują możliwości zadawania zapytań rekurencyjnych. Późniejsze standardy SQL – mimo, że przewidują możliwość zadawania zapytań rekurencyjnych w formie tranzytywnych domknięć, nie doczekały się jak dotąd pełnej implementacji. Nieliczne systemy zarządzania bazami danych dostarczające odpowiednie implementacje robią to albo w bardzo ograniczonej i niezgodnej ze standardami formie (zapytania hierarchiczne w Oracle), albo też w sposób zbliżony do opisanego w standardzie – jednak dowodząc w ten sposób głównie niedoskonałości ustandaryzowanych rozwiązań (formułowanie takich zapytań jest bowiem dość skomplikowane).

Dedukcyjne bazy danych – w których przetwarzanie rekurencyjne jest podstawą procesu dedukcji (ewaluacji zapytań) – nie doczekały się jak dotąd szerokiej akceptacji. W tym wypadku problem najprawdopodobniej nie wynika z braku implementacji (powstało ich wiele), lecz z braków samego języka w pozostałych obszarach zastosowań – w zależności od

wybranego dialektu Datalog'u, ograniczenia w zadawaniu zapytań mogą się różnić, jednak nie można powiedzieć, by był to język nadający się do typowych zastosowań bazodanowych.

Programowanie imperatywne również nie jest dobrym sposobem na zaspokojenie potrzeb związanych z zapytaniami rekurencyjnymi. Języki programowania wbudowane w systemy zarządzania bazami danych – jak PL/SQL czy Transact SQL często mają wbudowane ograniczenia dotyczące głębokości rekurencji, ponadto wszystkie tego typu języki zaimplementowane w komercyjnych bazach danych cierpią z powodu problemu niezgodności impedancji. Przetwarzanie rekurencyjne z poziomu aplikacji co prawda pozwala ominąć ograniczenia głębokości rekurencji, wiąże się jednak ze znacznym narzutem na komunikację z bazą danych. Dodatkowo, programowanie imperatywne ogranicza możliwości optymalizacji zapytań.

W sytuacji, gdy systemy zarządzania bazami danych oparte na SQL i Datalogu (oraz ich pochodnych) nie dają wystarczających możliwości przetwarzania rekurencyjnego – głównie ze względu na niedoskonałości wykorzystywanych języków zapytań i założeń teoretycznych, na których są one oparte, właściwym wydaje się zastosowanie innego – bardziej elastycznego języka. Praca ta proponuje wykorzystanie języka SBQL (Stack Based Query Language), opartego na podejściu stosowym (Stack Based Approach) do języków zapytań.

Praca przedstawia trzy podejścia do przetwarzania rekurencyjnego, zaimplementowane dla języka SBQL. Pierwszym są tranzytywne domknięcia, zaimplementowane w formie czterech wariantowych operatorów niealgebraicznych o różnej charakterystyce semantycznej i funkcjonalnej. Drugim są układy równań stałopunktowych – semantyczny odpowiednik zbiorów reguł Datalogu, choć o znacząco różnej składni. Trzecim są – omówione krótko – rekurencyjne procedury i funkcje oraz perspektywy, zaimplementowane w ramach wcześniejszych prac badawczo-rozwojowych w systemie ODRA.

Oprócz podstawowych informacji o składni i semantyce wszystkich trzech podejść, praca przedstawia również możliwości ich zastosowania oraz sposób budowy zapytań wykorzystujących te trzy podejścia – z wykorzystaniem przykładów opartych na wybranych możliwych obszarach zastosowań zapytań rekurencyjnych. Praca podejmuje próbę porównania tranzytywnych domknięć i układów równań stałopunktowych z punktu widzenia programisty na podstawie tych przykładów – szczególny nacisk jest położony na możliwość

dekompozycji problemów, pozwalającą programiście na łatwiejsze uporanie się z trudnymi zadaniami.

Jako że zapytania rekurencyjne są potencjalnie bardzo kosztowne do ewaluacji, praca przedstawia też możliwe do wykorzystania techniki optymalizacji zapytań. W podejściu stosowym najważniejszą grupą technik optymalizacyjnych są optymalizacje przez przepisywanie zapytań. Zapytania rekurencyjne również mogą być optymalizowane przez przepisywanie – dotyczy to zarówno tranzytywnych domknięć, jak i układów równań stałopunktowych – przedstawione jest zastosowanie tych technik do optymalizacji zapytań rekurencyjnych. Ponadto przedstawiona jest metoda optymalizacji układów równań stałopunktowych przez stratyfikację, jak również możliwość połączenia jej z optymalizacją przez przepisywaniem zapytań. Praca krótko omawia też najważniejszą z punktu widzenia wydajności wykonywania funkcji rekurencyjnych metodę optymalizacji przez usuwanie współdzielonych wyrażeń.

Praca omawia również najważniejsze możliwe obszary dla dalszych badań nad zapytaniami rekurencyjnymi w SBQL – reguły w stylu Datalogu, optymalizację zapytań rozproszonych, parametryzowane perspektywy oraz alternatywne modele składu. Wreszcie, praca przedstawia implementację tranzytywnych domknięć i układów równań stałopunktowym w systemie zarządzania bazami danych ODRA.

I. Introduction

1. Motivation

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. Extensive research has been conducted on processing Bill of Material and other hierarchical structures in relational databases— e.g. [1], [2], [3], [4], [5].

Recursion is also necessary for internal processing in computer systems, such as processing recursive metadata structures (e.g. Interface Repository of the CORBA standard), software configuration management repositories, hierarchical structures of XML [6] 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 the commonly used and implemented SQL standards (SQL-89 and SQL-92). The most advanced commercial DBMSs provide support for recursion, either through proprietary SQL extensions – e.g. Oracle [7] provides a language facility for “Hierarchical Queries”, or implementation of transitive closures using Common Table Expressions, as defined in SQL-99 – e.g. DB2 [8] and Microsoft SQL Server [9].

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 [10]. 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, no powerful programming abstraction capabilities, impedance mismatch during conceptual modeling of applications, poor integration with typical software environment (e.g. class/procedure libraries) and poor performance. Thus Datalog applications supporting all the features 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.

2. Transitive Closures, Fixpoint equation systems and recursion

It may be argued that transitive closure queries and fixpoint equation systems – in the form presented in this thesis – are not recursive queries. Recursion defines a simple basic case (or cases) and a set of rules that allow us to reduce all other cases to the basic case. Both declarative approaches presented in this thesis define a basic case and a set of rules that allows one to derive more complex cases – the exact opposite. They are also – in the approach presented here – calculated iteratively (although it could be calculated in a recursive way – it would be simply more expensive from the execution time point of view).

Both approaches, however, may be used (and are used) to solve the same class of problems as “real” recursive functions. As one of the main goals of this thesis is to provide a set of practical tools for database problems, it’s the author’s decision to treat them as recursive tools for the purposes of this thesis.

3. Key Work Objectives

The key work objectives for this thesis are:

- investigate the existing solutions in this area
- describe the syntax and semantics of all three recursive processing paradigms in relation to Stack Based Approach and SBQL
- investigate the aspects of these paradigms important from the implementation point of view, paying special attention to type checking and optimization of recursive queries
- investigate the relative qualities of all three paradigms, in order to provide suggestions on proper utilization of all three approaches to recursive queries.

4. Thesis Structure

First, this thesis discusses the state of the art in regard to recursive processing – all three paradigms (transitive closure queries, fixpoint equations and recursive functions) are discussed.

Then the thesis moves on to brief discussion of the Stack Based Approach (SBA) to query languages, and Stack Based Query Language (SBQL).

Next three sections discuss the three different paradigms to recursive processing in relation to SBA and SBQL. In each section we provide information on proposed syntax and semantics (including type checking where relevant), examples of queries based on those paradigms and a discussion of optimization methods which may be used for queries based on those paradigms.

This is followed by a section comparing those three approaches and discussing their preferred applications. Finally, brief discussion of possible further research and conclusions are presented.

II. 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.

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 the transitive closure problem, for instance, the 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 – as implemented in the database – may be insufficient.
- Some advanced closures, e.g. the query “get the total cost and mass of a given part X ” (assuming BOM structures) may be impossible to express assuming typical transitive closure operator syntax [11].

- Calculation of transitive closure leads to performance problems. The issue has resulted in many algorithms, such as those described in [3],[12] and [13].

SQL89 and SQL92 do not include transitive closure operator. Such operators – provided as vendor’s extensions to the language -are supported by Oracle and DB2 DBMS, and are included in SQL99 (aka SQL3) and SQL2003 standard proposals. In the following sections we present the transitive closures implementations provided by Oracle and DB2 DBMS-s.

i. Recursive Queries in Oracle

The Oracle DBMS provides support for recursive queries in the form of so called „hierarchical queries” [7]. 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 shown on Figure 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.

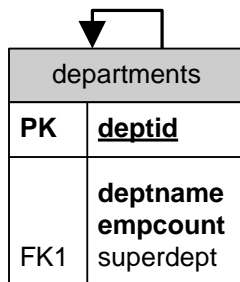


Figure 1 - simple database schema

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

```

SELECT sum(empcount)
FROM departments
CONNECT BY superdept = PRIOR deptid
STARTWITH deptname = 'Production';

```

Example 1 Transitive closure query in Oracle

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 Figure 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:

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

Example 2 Transitive closure with result formatting

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:

```
Production
--Engine assembly
--Chassis assembly
----Superstructure assembly
----Door assembly
-----Locks assembly
--Final assembly
```

Table 2 - Results from Example 2

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 lies in the evaluation order of a query with a join predicate. Materializing the join may 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) is 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) – the user has to rely on the pseudocolumns provided to him and nothing else. 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.

ii. Recursive Queries in SQL-99

SQL-99 (aka SQL3) was the first SQL standard proposal that introduces recursive queries, through a language facility named Common Table Expressions. A Common Table Expression allows the creation of a temporary, view-like construct, which can be recursively defined, thus allowing recursive processing without the need for stored procedures. 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.

Introduction of recursive queries in SQL-99 is in the author's opinion one of its 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 [14]. 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.

iii. 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. Transitive closure support in DB2 comes through the use of Common Table Expressions or CTE [8] – an implementation of language facility introduced in SQL3.

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):

```

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

```

Example 3 Transitive closure query in DB2

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 established 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 pseudocolumns containing additional information. However, DB2 provides another powerful facility for programmers. It allows the programmers to perform their own calculations in recursive queries. Information, such as a tuple's level in the hierarchy, can be calculated according to the user's need.

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.

For the table departments shown in Figure 1 and Table 1 consider the following query:

```

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;

```

Example 4 Transitive closure In DB2 – with result formatting

The query returns the following result set:

```
Production
--Engine assembly
--Chassis assembly
--Final assembly
----Superstructure assembly
----Door assembly
-----Locks assembly
```

Table 3 - Results for Example 4

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 with the SQL standard.

i. Recursive Queries in Microsoft SQL Server

Microsoft SQL Server 2005 introduced an implementation of Common Table Expressions introduced in SQL3 [9]. The query syntax for queries utilizing recursive CTE is as follows:

```
WITH cte_name ( column_name [,...n] ) AS
( initial_query UNION ALL recursive_query )

SELECT * FROM cte_name
```

The recursive evaluation proceeds as follows:

- CTE expression is split into anchor (initial_query, the starting point for transitive closure) and recursive queries
- anchor query is evaluated, returning the base result set (T_0)

- evaluate the recursive query, using T_i as input and T_{i+1} as output. Repeat evaluation until empty set is returned
- Return UNION ALL T_0 to T_n as result

Common Table Expressions in MS SQL Server are very similar (except for syntax details, which are irrelevant) to recursive queries defined in SQL3. Microsoft does not specify, what limitations are put on the recursive queries – one may reasonably expect, that they will be similar to those specified by SQL3. It is possible to limit the depth of recursion for a query, using a MS SQL Server facility named Query Hints – each INSERT, UPDATE, DELETE, or SELECT statement may have an additional OPTION clause, which allows user to specify additional instructions for the query optimizer and execution engine.

ii. Other transitive closure implementations

Transitive closure has been introduced not only as a feature in commercially sold DBMSs, but also as a research tool in prototype implementations. One of such proposals is presented in [15]. 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.

iii. Caching Transitive Closure Query Results

Transitive closure queries on relational databases are expensive. The cost analysis of transitive closure queries can be found in [5]. 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 [4]. In [16] 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 [17], 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 any silver bullets here.

2. Fixed Point Equations

Some recursive tasks cannot be easily expressed using transitive closure operation (although the difficulties will come from inadequacy of implementation rather than inherent lower expressiveness of transitive closure queries in general) , 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:

$$x_1 \leftarrow f_1(x_1, x_2, \dots, x_n)$$

$$x_2 \leftarrow f_2(x_1, x_2, \dots, x_n)$$

.

$$x_n \leftarrow f_n(x_1, x_2, \dots, x_n)$$

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.

i. 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 [18], 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, [10] presents the fixed point semantics of Datalog and [19] 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:

$$L_0 :- L_1, L_2, \dots, L_n$$

where each L_i is a literal of the form $p_i(t_1, t_2, \dots, t_k)$ 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:

parent(John,Mary)

Example 5 Datalog fact

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:

grandparent(X,Z) :- parent(X,Y),parent(Y,Z)

Example 6 Datalog rule

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. According to datalog researchers, the problem of infinite relations (but not oscillation) 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 using 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 \neg , 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 \neg , defining the properties of undergraduate students could be formulated as follows:

<code>und(X) :- stud(X), ¬grad(X)</code>
--

Example 7 Datalog \neg rule

(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

```
border="1">boring(chess) :- ¬interesting(chess)
```

Example 8 Datalog⁻ program with two minima models

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 the author's 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 author's 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 syllogisms 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, in principle, does not separate data and programs. Both are “rules” in the same abstraction frame. However, the fundamental differences between these two aspects concern the following factors: design of programs (databases are designed before programs that operate on them), client-server architecture (databases are on servers, programs are usually on clients); structural constructs (data structures are specifically constructed and constrained, especially in object-oriented databases; this does not concern programs); updating (data can be updated, while it makes little sense for programs). These fundamental differences cause that the unification of data and programs must meet severe limitations.
- Datalog implies significant problems with performance. Current optimization methods, such as magic sets, do not seem to be sufficiently mature and efficient.

ii. 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 [20]. 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 OOrelated) 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 [21],[22]. 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 [23] 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.

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 would have to be parameterized by side effects, which is a undesired property). 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 implemented in known commercial systems.

i. 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 author, 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 [7].

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.

```
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;

```

Example 9 Recursive procedure in PL/SQL

A sample `find_depts` procedure, working with the data model presented in Figure 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.

ii. 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 [24]. The language syntax and description is too complex to be included here. 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 [6]. 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:

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

```
]>
```

Table 4 - Data structure for Example 10

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:

```
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>  
};
```

Example 10 - Function Definition in XQuery

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

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

Example 11 - Function call for Example 10

Assume the following 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>

```

Table 5 - Input data for Example 10

For it the above query will return the following 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>

```

Table 6 - Results for Example 10

The recursive processing capabilities introduced in XQuery are an important part of the language, although the author is 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 function indicates otherwise.

In general, the critique of the XQuery solution concerns too exotic syntax with a lot of cryptic characters, tokens and constructs, which may be illegible for many users – although they are quite typical for XML-related language specifications coming from W3C. In this respect the solution is a step back in comparison to programming languages such as C++, Pascal or Java.

iii. 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 feature leads to clumsy code and problems with code writing and maintenance.
- Processing takes place outside DBMS, on the side of client application. It is more difficult to endanger the stability of DBMS with a poorly written function (for example one with 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 [25], 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 [23] (see 2.2.2), another - described in [15] (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 [26].

Evaluation models and optimization are also important fields. They have been covered in publications such as [27], 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 [28], again - with optimization methods. The authors of [29] propose recursive query optimization by transformation of logic programs, while the authors of [30] 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 [31] and [2].

Transitive closure queries are also a well-researched problem. The expressiveness of transitive closure queries is discussed in [32]- 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 [33]. Research into effective transitive closure query evaluation has also been conducted [34], [35].

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 [36] and [37].

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 with regard to 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.

III. Stack-Based Approach - an Overview

1. Query Language for Recursive Processing

Recursive processing facilities in a query language should be seamlessly integrated with other elements of that language. This is not always easy to achieve.

In some query languages – SQL and various languages derived from it – recursive processing facilities may be introduced, but the fact that the foundation on which the query language is built – be it a relational or object-oriented algebra, or a similar philosophy – does not support recursion, makes full integration of these facilities a difficult – if not impossible task. Recursive queries in such query languages will usually be very limited – often tailored to what the system developers expected as “typical applications”, the optimization will be non-existent or based on expensive techniques like materialization of transitive closures etc.

Other query languages, built solely around the concept of recursive processing may also miss the mark. Languages like Datalog provide very good recursive processing facilities, but often the languages do not provide adequate querying capabilities in other areas – the entire effort seems to be spent on making the recursion work.

In order to provide a query language with usable recursive processing facilities, a good starting point is necessary: a language, with theoretical foundations which include (or may be extended to include) both the concept of recursive processing and typical non-recursive queries.

Stack Based Query Language, based on the Stack Based Approach is an example of such query language. It is built around the concepts known better from programming languages – which makes introduction of recursive processing capabilities a relatively easy task, but also provides querying capabilities superior to those available to SQL programmers.

2. 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. For the purpose of this thesis the simplest data store model (called ASO [38]) is enough. In this data store model objects can contain other objects with no limitations on the level of the nesting of objects. Relationships between objects are also permitted. The presented approach can be easily extended to comply with other notions like classes, inheritance, dynamic roles, etc.

SBA store models, in particular ASO, 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 M_0 are triples defined below ($i_1, i_2 \in I$, $n \in N$, and $v \in V$).

- Atomic objects have form $\langle i_1, n, v \rangle$.
- Link objects have form $\langle i_1, n, i_2 \rangle$. An object is identified by i_1 and points at the object identified by i_2 .
- Complex objects have form $\langle i_1, n, S \rangle$, 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 Figure 2 and Figure 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.

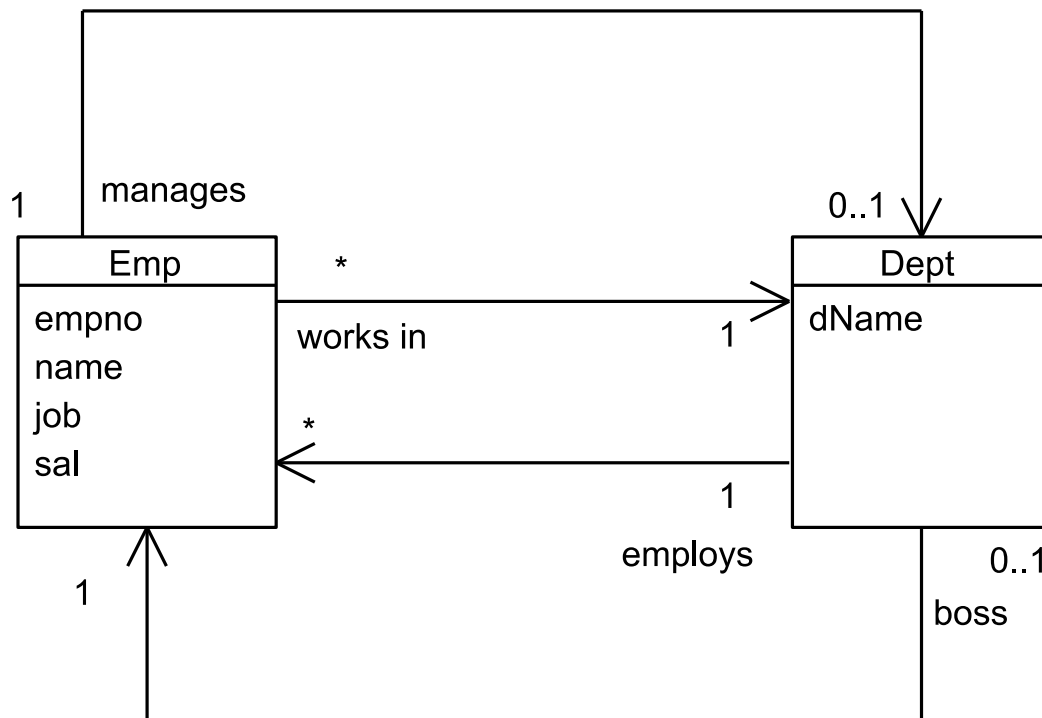


Figure 2 - Sample database schema

Objects:

```

    <i1, Emp, {<i2, empno, 123>, <i3, name, "Smith ">, <i4, sal, 3500>, <i5, job,
    "consultant ">, <i6, works_in, i20>} >
    <i7, Emp, {<i8, empno, 234>, <i9, name, "White ">, <i10, sal, 2900>, <i11, job,
    "developer ">, <i12, works_in, i20>} >
    <i13, Emp, {<i14, empno, 456>, <i15, name, "Blake ">, <i16, sal, 3200>, <i17, job,
    "manager ">, <i18, works_in, i20>, <i19, manages, i20>} >
  
```

```

<i20, Dept, {<i21, dName, "IT ">, <i22, employs, i1>, <i23, employs, i7>, <i24,
employs, i13>, <i25, boss, i13>} >
ROOT = {i1, i7, i13, i20}

```

Figure 3 - Example of SBA data store

3. 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.

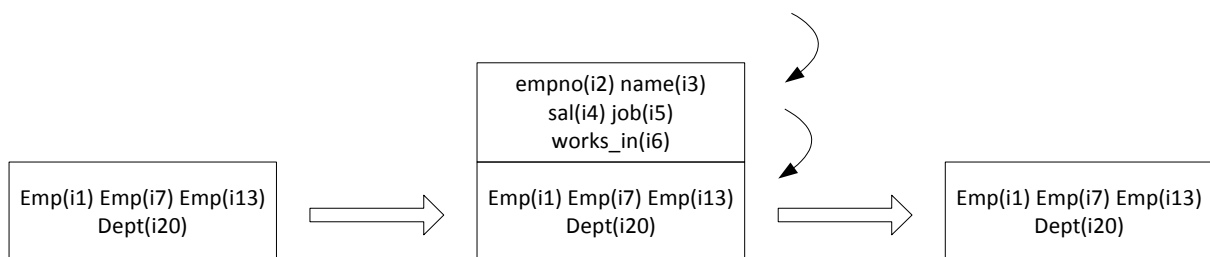


Figure 4 - Example of ES states during evaluation of a query

Some states of ES during evaluation of query Emp where $sal > 3000$ are depicted in Figure 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, $\text{nested}(\text{struct}\{x_1, x_2, \dots\}) = \text{nested}(x_1) \cup \text{nested}(x_2) \cup \dots$
- For other arguments the result of nested is empty.

Figure 4 contains the result of nested for i_1 (it returned a set: $\text{empno}(i_2)$, $\text{name}(i_3)$, $\text{sal}(i_4)$, $\text{job}(i_5)$, $\text{works_in}(i_6)$).

4. Stack Based Query Language (SBQL)

In this section we present the basis of the Stack Based Query Language (SBQL) [39], [40], [41], [42], [38]. The language has been first implemented in the LOQIS system [43], [44] 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 SBQL 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 constraints the operators are orthogonal.

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

i. Algebraic operators

An operator is algebraic, if its evaluation does not require the use of the environmental stack. The evaluation of the query $q_1 D q_2$ where D is an algebraic operator proceeds 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 D . 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.

ii. 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.

iii. Examples of queries in SBQL

Here we present examples of queries in SBQL addressing the example database shown in Figure 2

- Get employees earning more than 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))

iv. 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;
}
```

Example 12 - SBQL function

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)
```

Example 13 - SBQL function call

This query returns names and salaries of all employees of IT department that earn more than 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 [45].

IV. Transitive Closure Operator in SBQL

Transitive closure may be implemented as one of the operators in SBQL. Our proposal covers four variations on the transitive closure operator, along with their syntax, semantics, optimization methods and description of prototype implementation.

1. Transitive Closure Operators – syntax and semantics

i. The *close by* operator

The *close by* operator is the basic transitive closure implementation in SBQL. It is a non-algebraic operator and is defined as follows:

Syntax:

The *close by* operator is a non-algebraic binary operator. It's syntax is as follows:

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 AS1-AS3 models [38], 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. The operator described below is designed to solve this problem.

Alternative Semantics:

Semantics based on iteration, described above are not the only possible semantics that may be used for the close by operator.

A close by expression may be represented using fixpoint semantics. A q_1 close by q_2 query may be represented as an equivalent fixpoint equation:

$$x := q_1 \text{ union } x.q_2$$

where x is a fixpoint equation variable of the same type as results returned by q_1 and $x.q_2$ queries.

It is also possible to express the close by operator's semantics as a recursive function:

function closeby(x, q_2)

- result = bag(x)
- temp = evaluate $x.q_2$
- for each y in temp
 - result = result union closeby(y, q_2)
- return result

ii. The *close unique by* operator

The *close unique by* operator is designed to avoid the problem of potential infinite loops during calculation of transitive closure of a graph containing cycles. It is achieved by

elimination of repeated elements from the results. The process of elimination will result in increase of time required to evaluate the query, as each element returned by q_2 query has to be compared to the elements already in the T bag. Depending on chosen implementation method this may also result in increased memory consumption.

The operator is defined as follows:

Syntax:

The *close unique by* operator is a non-algebraic binary operator. It's syntax is as follows:

query ::= query close unique by query

Semantics:

The query q_1 **close unique 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 AS1-AS3 models [38], 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, the elements returned by q_2 are added to the elements in T only if there are no identical elements in T already.
 - 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

Alternative Semantics:

Semantics based on iteration, described above are not the only possible semantics that may be used for the close unique by operator.

A close unique by expression may be represented using fixpoint semantics. A q_1 close unique by q_2 query may be represented as an equivalent fixpoint equation:

$$x := \text{distinct}(q_1 \text{ union } x.q_2)$$

where x is a fixpoint equation variable of the same type as results returned by q_1 and $x.q_2$ queries.

It is also possible to express the close unique by operator's semantics as a recursive function – the function will be similar to the one defined for the close by operator, but will have to check, whether the currently processed element has already been visited (either by graph coloring or a collection storing all the visited elements).

iii. The leaves by operator

The *leaves by* operator is a variant of the transitive closure operator. This variant returns only those elements of the query results that did not lead to adding any new elements to the result bag – the leaves of the tree constructed by transitive closure operation.

It is defined as follows:

Syntax:

The *leaves by* operator is a non-algebraic binary operator. Its syntax is as follows:

$query ::= query \text{ leaves by } query$

Semantics:

The query $q_1 \text{ leaves by } q_2$ 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:
 - $\text{nested}(r)$ is calculated, the result is put on top of ENVS; in AS1-AS3 models [38], 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). If the q_2 returns an empty bag, the currently processed r element is marked as a member of the result bag – the way this information is stored may vary depending on the implementation decision.
 - 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. Elements marked as belonging to the result will now be selected from the contents of this section and returned as the result of the query.

Note that if q_2 returns a previously processed element, an infinite loop will occur. The operator described below is designed to solve this problem.

Alternative Semantics:

Semantics based on iteration, described above are not the only possible semantics that may be used for the close by operator.

A leaves by expression may be represented using fixpoint semantics. A q_1 leaves by q_2 query may be represented as an equivalent query utilizing a fixpoint equation:

(fixpoint{

$x := (q_1 \text{ as element, count}(q_1.q_2) \text{ as children) union } ((x.\text{element}.q_2 \text{ as element}),$
 $(x.\text{element}.q_2.q_2) \text{ as children})$

}).(x where children=0).element

where x is a fixpoint equation variable of the same type as results returned by q_1 and $x.q_2$ queries.

It is also possible to express the leaves by operator's semantics as a recursive function:

function leavesby(x, q_2)

- result = bag(x)
- temp = evaluate $x.q_2$
- if temp is empty return x ;
- for each y in temp
 - result = result union leavesby(y, q_2)
- return result

iv. The leaves unique by operator

The *leaves unique by operator* is a variant of the *leaves by operator*. This variant, like the *close unique by* variant is designed to avoid infinite loops while processing cyclic graphs.

It is defined as follows:

Syntax:

The *leaves unique by operator* is a non-algebraic binary operator. Its syntax is as follows:

$query ::= query \text{ leaves unique by } query$

Semantics:

The query q_1 **leaves unique 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 AS1-AS3 models [38], 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). If the q_2 returns an empty bag, the currently processed r element is marked as a member of the result bag – the way this information is stored may vary depending on the implementation decision.
- 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 elements returned by q_2 are added to the elements in T only if there are no identical elements in T already.
- 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. Elements marked as belonging to the result will now be selected from the contents of this section and returned as the result of the query.

Alternative Semantics:

Semantics based on iteration, described above are not the only possible semantics that may be used for the close by operator.

A leaves by expression may be represented using fixpoint semantics. A q_1 leaves by q_2 query may be represented as an equivalent query utilizing a fixpoint equation:

```
(fixpoint{
x :=distinct( (q1 as element, count(q1.q2) as children) union ((x.element.q2 as element),
(x.element.q2.q2) as children))
}).(x where children=0).element
```

where x is a fixpoint equation variable of the same type as results returned by q_1 and $x.q_2$ queries.

It is also possible to express the leaves unique by operator's semantics as a recursive function – the function will be similar to the one defined for the leaves by operator, but will

have to check, whether the currently processed element has already been visited (either by graph coloring or a collection storing all the visited elements).

2. Transitive Closure – type checking

The postulated type system for SBA is a semi-strong type system described in [46]. Like all other queries, transitive closure queries need rules for typechecking.

The result of a transitive closure is calculated as a union of the result of query q_1 and results returned by subsequent evaluations of query q_2 . The decision tables for transitive closure are identical as those defined for the union operator. The processing is different, however – since the transitive closure is a non-algebraic operator, after typechecking q_1 the result of nested function on signature of q_1 's result is put on the static environment stack, then q_2 is typechecked in the new environment.

The decision tables are as follows:

q_1	q_2	q_1 close by q_2
τ	τ	τ
Other		error

τ - metavariable

Table 7 - decision table for signature base

q_1	q_2	q_1 close by q_2
$[x1..x2]$	$[y1..y2]$	$[x1+y1..x2+y2]$

$x1, x2, y1, y2$ - metavariable

Table 8 - cardinalities (card attribute)

q_1	q_2	q_1 close by q_2
-	-	Bag
-	Bag	Bag
Bag	-	Bag
Bag	Bag	Bag
Other		Error

Table 9 - type of result collection (collectionKind attribute)

q_1	q_2	q_1 close by q_2

-	-	-
η	η	η
Other		error

η - metavariable

Table 10 - name of the result type (typeName attribute)

The decision tables are identical for all four incarnations of the transitive closure operator.

3. Transitive Closure – stop problem

There is a problem inherent to all recursive calculations - including the transitive closure operator evaluation. In certain conditions, the evaluation of a query may not stop, but go on forever (in practice – until the stack overflows and the system crashes).

Many systems offering recursive processing capabilities try to solve this problem by limiting the recursive queries in some way e.g. Datalog limits on the negation operation. While this approach may be useful in a language, which offers very limited querying capabilities, in the case of a more powerful language – like SBQL – it will not be a desirable approach. The number of operations, which may lead to an infinite evaluation loop is simply too great, limiting access to them would result in many potentially useful queries being banned.

Let us consider the following query:

```
(1 as x) leaves by
((a/x + x)/2 as x)
```

Example 14 Square Root calculation

The query above is a transitive closure implementation of square root calculation. If a is replaced with a number, this query will calculate approximations of its square root – a more precise approximation in each step. It will not, however, return a result, as the transitive closure calculation will never end – there’s no stop condition in this query, and a more precise approximation can always be found (at least if we ignore the floating point precision problem).

In order to stop users from formulating such queries, even the arithmetic operators would have to be banned in transitive closure queries – even, though the query may be easily modified to a form, which will stop after an arbitrary number of steps:

((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)

Example 15 Square Root calculation with a stop condition

This variant of the square root calculation query uses a named value *counter* to limit the number of iterations to 5 (of course the number may be set to any desired value, to achieve greater precision).

The Author considers any attempt to stop users from asking queries which will result in an infinite evaluation loop to be futile and harmful. Such attempts will also stop them from asking many potentially useful queries, and will not remove the danger entirely, as a determined user will be capable of finding a query causing an infinite loop anyway.

On the other hand, the danger to DBMS's stability caused by such queries is evident. The DBMS has to be protected from such queries in a different way. Our suggestion is to utilize a well-known method of limiting the resources available to process a single query – if a query may utilize only a specific amount of time or memory before being stopped by the DBMS as potentially dangerous, the goal of protecting the DBMS stability without limiting the recursive queries will be achieved. Such a system, if designed with flexibility in mind, could e.g. allocate different amount of resources to ad-hoc queries (which may be dangerous) and to queries defined as views (which can be verified by the database administrator as being too resource consuming, but not presenting a danger of an infinite evaluation loop).

V. Transitive Closure operator in SBQL – usage examples

Transitive closure operators may be utilized to formulate queries, which are impossible or very difficult to formulate in other query languages - such as SQL in one of its many incarnations.

In this section we'll consider two of the many possible applications for transitive closure operations. The first one is a Bill of Material problem – querying a data structure representing components of a complex industrial product. The second is a workflow system – examples show, how the transitive closure may be useful for querying a workflow metamodel.

1. Bill of Material

Bill of Material problems are – like genealogical trees and company organizational hierarchies – among the standard examples of usefulness of recursive processing. It is not surprising, as Bill of Material is an interesting problem from a business point of view, but is – unfortunately – poorly supported by relational DBMSs.

In a Bill of Material problem we deal with a situation, in which a certain product is described as a sum of its components – which, in turn, consist of other components and so on – until the level on which we deal with simple, *detail* parts is reached. In this context we may be interested in information such as the product's (or component's) weight, manufacturing costs and other similar data useful from business or technical point of view, which may only be obtained as a result of recursive calculations.

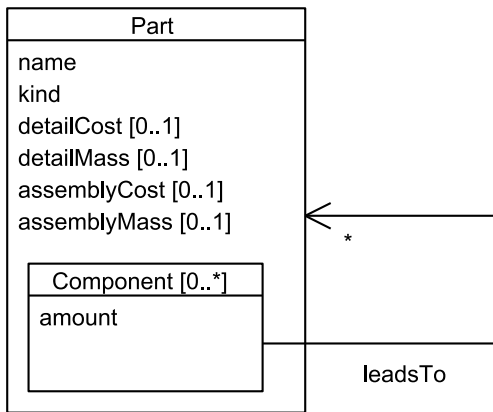


Figure 5 - BOM schema

Let's consider a possible data model for a simple Bill of Material problem. The system stores information about *parts*. 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* and *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.

The most basic transitive closure query we can formulate is retrieving information about components, which are used to create a specific part:

(Part where name="engine")

close by

Component.leadsTo.Part

Example 16 Simple transitive closure over BOM schema

The query's design is quite simple – the first subquery (Part where name="engine") selects the starting element – in this case element named "engine". The second subquery specifies, how to reach the other elements constituting the transitive closure results – in this case we're interested in other Part elements that may be reached by following the leadsTo pointer object of the Component sub-object.

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:

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

Example 17 Transitive closure over BOM schema

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:

```
((((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)))
```

Example 18 Complex transitive closure over BOM schema

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

This query could be further refined, in order to remove all aggregate parts (so only the detail parts will be returned 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:

```

((((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)))

```

Example 19 Complex transitive closure over BOM schema

- Returning objects which do not have components:

```

((((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)))

```

Example 20 Complex transitive closure over BOM schema

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

```

((((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)

```

Example 21 Complex transitive closure over BOM schema

2. Workflow

Workflow systems are – like Bill of Material – an important tool in modern business. The ability to analyze or execute a business process described as a workflow is very important from business point of view.

The potential of Stack Based Approach in context of workflows has already been researched and described in [47]. The discussion below does not have as ambitious and far-reaching goals as that paper – the author’s intention is to show the potential usefulness of transitive closures in workflow applications, using greatly simplified (for the sake of clarity) examples.

Let’s consider a part of a possible workflow metamodel. An *Activity* has a *name*, *type* and *state*, as well as collections of *pre-* and *post- conditions* and *constraints*. Execution of an activity takes no more than *maxTime*. Activities are connected with each other via *Transitions*. Transitions may specify *conditions*. Activities may also require *Resources* (which have *names*), the *amount* of resources required by each activity is specified in *RequiredResources*.

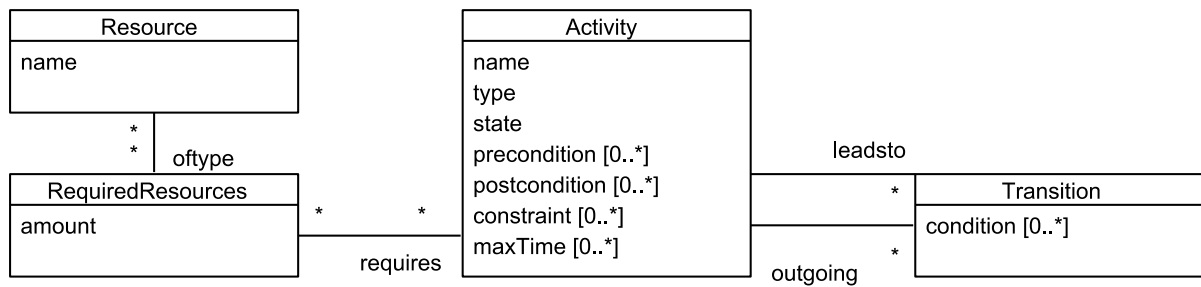


Figure 6 – Simple workflow metamodel

The simplest transitive closure SBQL query over this schema finds all activities that may be reached via transitions from a certain activity – e.g. one named “Planning”.

(Activity where name = "Planning")
close by
(outgoing.Transition.leadsto.Activity)

Example 22 – Simple transitive closure query over workflow metamodel

This query first selects activities having with *name* equal to “Planning”. The transitive closure relation is described by the subquery *(outgoing.Transition.leadsto.Activity)*. It returns all *Activity* objects which may be navigated to via the pointers that lead to and from appropriate *Transition* objects.

As we know the maximum time for execution of each activity, we may want to calculate the maximum amount of time that may be required to complete a given business process. Let’s assume that the given process starts with „Planning” and all activities that may be reached via transitions from this activity are parts of this process.

First we augment the above query so that it returns pairs. Each such pair constructed by means of the dependent **join** consists of an activity and the time needed to complete this activity counted from the beginning of the whole workflow.

```

(
  ((Activity where name="Planning") as x)
  join
  (x.maxTime as howLong)
)
close by
  
```

```
(
  (x.outgoing.Transition.leadsTo.Activity as x)
  join
  ((x.maxTime + howLong) as howLong)
)
```

Example 23 Transitive closure query over workflow metamodel

The query uses a named value in order to calculate the time. The maximum time the initial activity may require is named *howLong* and paired with the initial activity. In subsequent iterations the *howLong* value from parent object is used to calculate the value (that will be assigned the same name) to be paired with child elements.

With the above query in hand we can compute the maximum length of the workflow. We simply restrict the result of this query to the part *howLong* and calculate the maximum:

```
max( ( ((Activity where name="Planning") as x)
  join
  (x.maxTime as howLong)
)
close by
(
  (x.outgoing.Transition.leadsTo.Activity as x)
  join
  ((x.maxTime + howLong) as howLong)
)
.howLong
)
```

Example 24 Transitive closure query over workflow metamodel

This query computes the time needed to complete the critical path but does not return the activities along it. Such information is very important in most cases. SBQL does allow collecting this information. Another named variable is needed. This variable will store the collection of activities along the particular path. Here is the query which collects the path items:

```
(
  ((Activity where name="Planning") as x)
  join
```

```

        ((x.maxTime as howLong), x as path)
    )
    close by
    (
        (x.outgoing.Transition.leadsTo.Activity as x)
        join
        ((x.maxTime + howLong) as howLong, (path ∪ x) as path)
    )

```

Example 25 Transitive closure query over workflow metamodel

Now, we can use this query in order to select the critical path. We just group the result of the above query under the name *paths*. Then we find the path with the longest completion time (i.e. the value of the variable *howLong*):

```

((
    (
        ((Activity where name="Planning") as x)
        join
        ((x.maxTime as howLong), x as path)
    )
    close by
    (
        (x.outgoing.Transition.leadsTo.Activity as x)
        join
        ((x.maxTime + howLong) as howLong, (path ∪ x) as path)
    )
) groups as allPaths)
.((allPaths as result) where result.howLong = max(allPaths.howLong))
.result
.path

```

Example 26 Complex transitive closure query over workflow metamodel

The results of recursive queries may be also used as a starting point for other calculations – for example, the type and amount of resources required by a specific business process.

```

( (Activity where name = "Planning")
close by
(outgoing.Transition.leadsTo.Activity)).

```

```
( (requires.RequiredResource.  
(oftype.Resource.name as res,amount as howMany)) group as requiredResources).  
(distinct(requiredResources.res) as resource).  
(resource, sum((requiredResources where res=resource).howMany))
```

Example 27 Complex transitive closure query over workflow metamodel

This query first finds all the activities that are a part of a business process, then finds all resources required in various activities pairing them with the required amounts, groups all those pairs in a collection named *requiredResources* and processes the collection in order to merge pairs with duplicate resource names.

VI. Optimization of Transitive Closure

Transitive closure queries require optimization in order to become truly useful. Otherwise the relatively high cost of evaluating queries that utilize transitive closure operators will seriously limit their potential applications.

Below we discuss the techniques that may be utilized to optimize transitive closures. Those techniques are generic, and may be used also with other non-algebraic operators. They may be divided into two groups – query rewriting and query result materialization.

A more elaborate discussion of methods for query optimization by rewriting in Stack Based Approach is available in [48]. In this chapter we discuss only the application of those methods to recursive queries.

1. Optimization by rewriting queries

Two most useful methods that may be used for transitive closure query optimization are the factoring out independent subqueries and pushing out selections before the transitive closure operator. Short discussion of those methods is presented below, along with the description of general framework for query optimization used in SBA.

i. Query optimization in SBA – general framework

Some queries in SBQL may be evaluated outside loops implied by non-algebraic operators. If a query has the following construction:

q_1 **non-algebraic operator** q_2

the query q_2 will be evaluated once for each element returned by q_1 . A part of the query q_2 , however, may be independent from q_1 – the result of evaluation of q_2 will be the same if it is evaluated within the context of q_1 and outside its context.

The independence of q_2 or its part from q_1 is decided by the analysis of name binding.

Every non-algebraic operator opens its own scope on the environment stack. Each name that occurs in a query is bound on one of the levels of that stack. It is not important – for

optimization purposes – on exactly which level of the environment stack a name is bound. The important distinction is, whether the level was on the environment stack before the evaluation of the query in question commenced. We divide all the names in a query into two groups:

- *free names* – names bound in the *bottom scopes* (scopes that were on the environment stack before the evaluation of the query commenced)
- *non-free names* – names that are not free

To determine, on which level a name is bound, the query is analyzed statically. During the static analysis:

- each non-algebraic operator is assigned the number of the scope it opens
- each name is assigned two numbers:
 - the *stack size*: number of environmental stack scopes open when the binding of this name is being performed
 - the *binding level*: the number of stack scope in which the name is bound

Those numbers are relative to the bottom scopes of the query – the free names are bound on level 1, non-free names on levels greater than 1.

ii. Factoring out independent subqueries

Factoring out independent subqueries is a generalization of a technique used in optimization of nested queries in the relational model. The idea of factoring out independent subqueries, described in [48] deals with unnecessary repeated evaluation of some subqueries during the evaluation of a non-algebraic operator.

Let's consider a query containing a non-algebraic operator:

q_1 **non-algebraic operator** q_2

If all names in a subquery of q_2 – we'll name it q_{2a} – are *free names*, the result of evaluation of such subquery do not depend on the scope opened by the non-algebraic operator. In every step of the loop evaluating the non-algebraic operator the result of q_{2a} evaluation will be the same, so calculating the result of q_{2a} only once and re-using this evaluation result in

every step of evaluation of the query q_2 will result in shorter evaluation time for the entire query.

The factoring out operation is executed as follows:

- the optimizer detects a subquery (part of the AST) that contains only names that are free names
- a new unique auxiliary name is introduced to name the results of this query
- the independent subquery is named with this name, put before the query and connected to it by a projection operator (to store the result on the environment stack).
- the auxiliary name is put in the original place of the independent subquery

After factoring out the independent subquery, our query has the following structure:

$(q_{2a} \text{ as } \textit{uniquename}).(q_1 \text{ non-algebraic operator } q_2')$

Where q_2' is the query q_2 in which subquery q_{2a} is replaced with *uniquename*. Note, that if q_1 **non-algebraic operator** q_2 is a part of a larger query, we're factoring out the q_{2a} subquery only in front of it, not outside the entire query.

Let's consider an example (the query uses the Workflow data model presented in Figure 6):

```
(Activity where name = "Planning")
close by
(outgoing.Transition.leadsTo.Activity where maxTime<avg(Activity.maxTime))
```

Example 28 – Transitive query with an independent subquery – before factoring out

The $avg(Activity.maxTime)$ subquery has some interesting properties. First – it can be quite expensive to calculate (depending on the number of activities in the database etc). Second – it will be calculated for every activity processed by the transitive closure operator (again, potentially many times). Third – it is independent of the *close by* operator.

This means, that it is possible to factor out this subquery, in order to optimize the entire query. After factoring out, the query will have the following form:

```
(avg(Activity.maxTime) as x).((Activity where name = "Planning"))
```

close by

(outgoing.Transition.leadsTo.Activity where maxTime<x)

Example 29 – Transitive query with an independent subquery – after factoring out

After factoring out, the subquery will be executed just once – and its results will be reused in every iteration.

iii. Pushing out selections before the transitive closure operator

Pushing out selections before a non-algebraic operator is another query rewriting technique available in SBA. This technique, described in [48] utilizes the distributivity property of some non-algebraic operators.

Distributivity property

A non-algebraic operator θ is *distributive*, if for any syntactically correct queries q_1, q_2, q_3 (q_1 and q_2 are union-compatible) and for any database and environmental stack holds the following:

$(q_1 \text{ union } q_2) \theta q_3$

Is semantically equivalent to

$(q_1 \theta q_3) \text{ union } (q_2 \theta q_3)$

In other words, a non-algebraic operator θ is distributive, if for any query

$q_1 \theta q_2$

its results may be calculated as an union of all results of

$r \theta q_2$

where r is one of objects returned by the query q_1 – thus the result is always a union of partial results for all objects returned by q_1 .

Operators **close by** and **leaves by** are distributive – as can be clearly seen in the description of their semantics, both return the union of partial results for all objects returned by q_1 .

Operators **close unique by** and **leaves unique by** are not distributive – in order to make each element of the result set unique, some objects may be removed from it, thus the result won't be always a union of partial results.

Pushing out selections

Generalization of the pushing out selection before a join technique known from the relational DBMSs, this method is another variant of the independent subqueries optimization method.

If a selection predicate is independent of a distributive non-algebraic operator, then the selection is not factored out, but pushed before that operator. If it is not possible to push out the entire selection predicate, sometimes it is possible to push out a part of it – if the predicate is not independent as a whole, but has the form:

p_1 **and** ... **and** p_{k-1} **and** p_k **and** p_{k+1} **and** ... **and** p_n

and some subpredicate p_k ($k \in \{1, 2, \dots, n\}$) is independent of that operator, then the selection with that subpredicate is pushed and predicate is rewritten to the form:

p_1 **and** ... **and** p_{k-1} **and** p_{k+1} **and** ... **and** p_n

Let's consider the following example: assume, that our BOM data model has another class, called Car. Class Car – among other attributes – has an attribute model (containing the name of the car model) and a link named Components to the class Part – leading to the top-level part, representing the entire car assembly. A user familiar with other query languages might want to ask the following query:

```
Car.Components.Part  
close by  
Component.leadsTo.Part  
where  
model = "Testarossa"
```

Example 30 Query with a predicate independent of the close by operator

in order to find all components used to create a car named Testarossa. Obviously, the query – if formulated in this way – would be very expensive to calculate, as the BOM hierarchies for other car models would also be calculated.

Fortunately, the predicate

model = "Testarossa"

is independent of the close by operator, thus it can be pushed in front of it – as a result we have the following, semantically equivalent query:

(Car where model = "Testarossa").Components.Part
close by
Component.leadsTo.Part

Example 31 The same query after optimization

Now, only the BOM hierarchy for the Testarossa model will be traversed by the query, thus significantly reducing the time required to process the query. It is worth noting, that this optimization technique will be mostly used for queries written by users less familiar with SBQL and more with query languages like SQL. Proficient SBQL users will be more likely to write a query that does not require such optimization.

2. Other methods for Transitive Closure query optimization

Query rewriting is not the only possibility for query optimization available. The other available methods involve caching of the results and materialization of transitive closure. Those methods require additional resources (memory or disk space), but may be useful if the same (or similar) query is executed often. In our opinion, the first technique described below is not well suited for SBQL, and is presented only for the sake of completeness.

i. Query result caching

If a specific query is executed often (this may apply to a subquery, not only to a whole query asked by the user), it may be possible to store the results of this query and use them when the query is asked again instead of evaluating the query once more.

Such caching would require the database management system to store a list of queries for which the query results are cached. Just like in the case of query optimization using indices, the optimizer would replace the query with retrieval of the cached results.

The cached query results may be invalidated by update operations. Such operations should be detected by the database management system and the query removed from the cached query index. The database management system has to be able to determine, which operations will alter the query result. Note, that even if an object (or object's attribute) was not a part of the original result, an update to it may alter the query result – e.g. if the update causes the object to satisfy the original query.

This technique is often implemented as means of optimization for relational database management systems extended with transitive closure facilities. Simplicity of SQL makes it relatively easy to determine, whether an update operation invalidated a result set or not, and whether a query may use one of the cached results.

The expressiveness of SBQL makes development of an algorithm capable of determining, whether any given update operation will invalidate the cached results of a query impractical (if not impossible) – unless we're willing to treat any modification of an object with the same type as the cached results as invalidating the cached results.

Another problem is the number of queries, which should be cached at the same time. If the number is too small, the cache may not fulfill its purpose well enough - if the system handles a large number of queries, some cached queries may be replaced with new ones before they are used again. If the number is too big, the resource consumption may become an issue, and the optimizer has to compare too many subqueries with the index positions.

This makes the query result caching a technique, which is appropriate only for specific situations – large sets of data, which are queried often with a small set of queries, but modified rarely. The next described technique – materialization of transitive closure via views – will be better suited for such cases. If the updates of the queried data are frequent, the costs related to maintenance of cached query index and its utilization may outweigh the benefits.

ii. **Materialization of transitive closure**

The query result caching method depends on caching all query results and hoping, that they may be re-used. Even if we limit the result caching to queries containing certain operators (e.g. transitive closure) or taking more than specified time to evaluate, not all queries will be asked again before their results become invalidated.

There is, however, a way to increase the chance of reusing the query results. Database administrator will usually be able to determine, which queries will benefit from such reuse. His information may come from the system logs, users or application designers – in any case, it will be more accurate than any assumptions made by the designers of the optimization engine.

In order to utilize this information, the database administrator may use materialized views. A query that is likely to be used often enough may be encapsulated within a view, the view may then be materialized – if the database engine provides support for view materialization.

While this approach may seem similar to the previous one, there are some significant differences:

- The views will be referenced to directly in the queries. This will remove the need for the optimizer looking for matching subqueries
- The number of cached queries is no longer an issue – system's administrator may select any arbitrary set of views to materialize depending on the needs and capabilities of the system
- The administrator may choose the queries to be materialized to match the desired profile – often asked queries, expensive to evaluate, based on rarely updated objects.

The users have to be aware of the existence of the materialized view to take advantage of it – unlike in the case of cached queries. Overall, however, materialized views are in author's opinion a better approach than query caching.

3. **Summary**

The optimization techniques presented here are not limited just to the optimization of transitive closure queries. Various non-algebraic operators in SBQL operate in very similar

manner, making operators that in other query languages have to be considered as special cases just another example of a large group of operators for which the same optimization techniques may be utilized.

VII. Fixpoint Systems in SBQL

SBQL's flexibility provides the potential necessary to support many different programming paradigms – including paradigms that in the past have inspired the creation of completely new programming and query languages. One of the paradigms, which may be supported is the deductive database concept that is the basis of the Datalog query language.

There are several methods which may be used to evaluate Datalog programs. One of them, especially interesting as a method for recursive processing in SBQL, is treating a Datalog program as a system of fixpoint equations.

1. Fixpoint Systems – Syntax and Semantics

i. Syntax

Fixpoint is a statement that groups one or more fixpoint equations, each consisting of a variable declaration (name and type) and a query used to calculate the value of this variable:

```
fixpoint {  
  
variable name1 : signature1 := query1;  
  
...  
  
variable namen : signaturen := queryn;  
  
}
```

ii. Semantics

The proposed fixpoint system's syntax differs from the syntax used by Datalog-like languages. Instead of having a set of separate declarations of deductive rules, we propose to group the entire set of rules as a single query.

The query **fixpoint** {...} is evaluated as follows:

- A new scope is opened on the environment stack

- Each declared variable is initialized with an empty value appropriate to the type defined for it, the variable is then put on the environment stack
- The following steps are carried out in loop:
 - Each of the queries is evaluated, the results of the queries stored under unique names on the result stack
 - Result of each query are compared to the corresponding variable stored on the environment stack. The result of this comparison are stored, then the results are assigned as the new value of the variable
 - If all results were equal to the value of corresponding variables on the environment stack, the loop is ended. Otherwise the next iteration of this loop starts
- The variable values are stored on the result stack, the scope opened by the fixpoint statement on the environment stack is closed.

The queries may refer to variables defined by other equations within the fixpoint system – just like Datalog rules refer to other rules. The evaluation of a fixpoint system ends, when the fixed point is reached – another iteration will not change the value of any of the declared variables.

iii. Fixpoint Systems and persistence

Fixpoint systems in SBQL are not persistent – unlike Datalog, in which deductive rules once declared are treated as a persistent information in the database. A fixpoint system is treated in exactly the same way any other query would be. This, however, does not mean that a fixpoint system cannot be made persistent. A fixpoint system may be used as a seed for a view (standard or materialized) or returned by a procedure – they may allow the user not only to make the user to make the fixpoint system persistent, but also to e.g. parametrize it. There is no need for a separate facility that would allow users to create persistent fixpoint systems.

2. Fixpoint Systems – type checking

Like other queries, a fixpoint system has to be typed. Type checking of fixpoint systems is a much more complex problems than in the case of other queries. Since the type of each

equation's result depends not only on the equation itself, but also on the other equations in the system, the signature of equation's result may also change in each of iterations. This means that the approach to type inference used in other SBQL queries cannot be used in fixpoint systems.

iv. Automated type inference

It is possible to introduce automatic type inference for fixpoint systems. To determine the correct approach let us first to consider a fixpoint system (we omit the variable signature declaration in all examples in this section) with only a single equation, e.g.:

```
fixpoint {
  activities := (Activity where name="Planning") union
               (activities .outgoing.Transition.leadsTo.Activity); }
```

Example 32 Simple fixpoint system

If we assume the activities variable to be of *any* type for the time being, then we can reduce the equation for type inference purposes to the following equation:

```
activities := (Activity where name="Planning");
```

Example 33 fixpoint equation for the purposes of type inference

We can determine the signature of this equation. It is $i_{Activity}[card=0..*]$. In the next step, we have to verify, that the inferred type is correct. In order to do this, we use the original equation and determine its signature, using the signature inferred in the previous step for the *activities* variable. If the signature calculated in this step is the same as the signature inferred in the previous step, then the inference is correct.

When a fixpoint system consists of more than one equation, additional steps are necessary. Let's consider a simple fixpoint system that consists of three equations:

```
fixpoint {
  start := ((Activity where name="Planning") as x) join ((x.maxTime as howLong));
  path  := start union ( (path.x.outgoing.Transition.leadsTo.Activity as x)
                        join ((howLong + x.maxTime) as howLong));
  final := max(path.howLong)
}
```

Example 34 Fixpoint system with three equations

If we assume all three variables in those equations to be of *any* type, and try to infer the signatures of all three equations simultaneously, we encounter a problem. Although the inference of the *start* variable's signature works correctly, the *path* and *final* variables' signatures will be inferred incorrectly. In order to infer the *path* equation's signature we need the "real" (already inferred) signature of *start* equation, and to infer the *final* equation's signature we need the "real" signature of *path* equation. To solve this problem we need first to perform stratification of the fixpoint system. This particular fixpoint system will be divided into three strata:

```
start := ((Activity where name="Planning") as x) join ((x.maxTime as howLong));
path  := start union ((path.x.outgoing.Transition.leadsTo.Activity as x)
                    join ((howLong + x.maxTime) as howLong) );
final := max(path.howLong)
```

Example 35 Stratified fixpoint system

Now, we may proceed with the inference, starting with the lowest stratum (1). Each stratum will be processed separately. The results of type inference in a lower stratum will be used to infer the types in a higher stratum. Stratification, however, not always solves the problem. Consider the following fixpoint system:

```
fixpoint {
odd   := (Activity where name="Planning") union
        (even .outgoing.Transition.leadsTo.Activity);
even  := (odd .outgoing.Transition.leadsTo.Activity);
}
```

Example 36 Fixpoint system with two equations in the same stratum

The *odd* and *even* equations depend on each other, so they will be placed in the same stratum. The first iteration of the inference algorithm will not produce correct results. The correct result for this (and similar) fixpoint system could be achieved in one of the subsequent iterations. However treating fixpoint system's type inference as a de facto another fixpoint system is dangerous. Some evaluations may never reach the fixpoint, e.g.:

$x := (1 \text{ as } a) \text{ union } (x \text{ as } a)$

Example 37 Fixpoint system with infinitely nesting binders

This equation generates infinitely nesting binders. The possibility of a type checking system falling into infinite loop is unacceptable. Thus, a fixpoint-style signature inference cannot be used. In order to solve this problem, we may give the user the possibility to explicitly define types of equations. The fixpoint definition becomes as follows:

fixpoint $\{x_1 : \text{sig}_1 :- q_1; x_2 : \text{sig}_2 :- q_2; \dots x_m : \text{sig}_m :- q_m\}$

where:

x_1, x_2, \dots, x_m are names of variables in this equation system,

$\text{sig}_1, \text{sig}_2, \dots, \text{sig}_m$ are optional signature definitions,

q_1, q_2, \dots, q_m are SBQL queries with free variables x_1, x_2, \dots, x_m ;

This approach allows explicitly defining the equation signatures in situations in which they are too difficult for the type inference algorithm to determine, while allowing the user to omit the definition for simpler ad-hoc queries. The type inference algorithm works as follows:

- Stratify the equation system
- Process each stratum, starting from the lowest:
- Assume variables in this stratum to be of type *any* unless explicit type is provided
- Infer the equations' signatures using this assumption
- Verify the inference
- If the verification is not successful, report a type checking error

This approach is an engineering compromise between full automatic type inference – which is expensive and may (as Example 37 shows) fall into infinite loop - and not using type inference at all.

The type inference, however, would be useful only in the case of less complex fixpoint systems, the more complex ones would still require user to define return types. In author's

opinion, it is better to require the user to declare types for all fixpoint system variables and not use automatic inference at all. Such approach has the following advantages:

- The consistency of this approach will be less confusing to the user than an approach in which some fixpoint equations would require the declaration of variable types while others would not
- Skipping the attempt to infer types automatically will reduce the time required to evaluate a fixpoint system
- The user is aware of the type of data returned by the fixpoint system. This means, that he'll be able to utilize the fixpoint results in queries or procedures with greater efficiency.

v. **Fixpoint systems type checking – recommended approach**

In our approach, each variable in the fixpoint system has its type explicitly declared. The declared type will be used for the following purposes:

- Determining the type of the fixpoint system result
- Correct initialization of the fixpoint system variables
- Type checking of the fixpoint equations

As each of the fixpoint system variables' signature is declared, it is easy to determine the type of the fixpoint system result. For the fixpoint system:

fixpoint{ $x_1 : sig_1 := q_1; x_2 : sig_2 := q_2; \dots x_m : sig_n := q_n$ }

the result type will be:

struct($sig_1, sig_2, \dots sig_n$)

The initialization of fixpoint system variables is an important problem – the fixpoint system variables have to be initialized with values, which will ensure the proper behaviour of the queries in the initial iteration.

If the cardinality assigned to a variable is different than 1..1 or the type is not a simple type, the variable should be initialized as an empty *Bag* with the declared type and cardinality. If,

however, the type is a simple type with declared cardinality of 1..1, the variable should be initialized with a value corresponding to the variable's type:

- *float* and *integer* variables should be initialized to 0
- *string* variables should be initialized to an empty string

The fixpoint equation's signature should be identical to the signatures declared for corresponding fixpoint system variable's. The decision tables are as follows:

Variable	Query	Variable := query
τ	τ	τ
Other		error

τ - metavariable

Table 11 - decision table for signature base

Variable	Query	Variable := query
[x1..x2]	[x1..x2]	[x1..x2]
Other		error

x1, y1 - metavariable

Table 12 - cardinalities (card attribute)

Variable	Query	Variable := query
-	-	-
Bag	Bag	Bag
List	List	List
Other		Error

Table 13 - type of result collection (collectionKind attribute)

Variable	Query	Variable := query
-	-	-
η	η	η
Other		error

η - metavariable

Table 14 - name of the result type (typeName attribute)

VIII. Fixpoint Systems in SBQL - usage examples

One of the possible reasons for using fixpoint systems in SBQL is making the recursive queries more readable. The more advanced examples of transitive closure queries were quite complex, queries of such complexity may prove to be difficult both to write and analyze. A way to utilize the fixpoint system in order to construct a recursive query with an easily understood structure is presented below.

1. Bill of Material

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.5):

```
fixpoint {  
  parts : Part[1..*] := (Part where name="engine") union  
    (parts.Component.leadsTo.Part);  
}
```

Example 38 Simple fixpoint system with a single equation

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.

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

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

Example 39 Fixpoint system over the BOM schema

Only variable final is returned as the query result. The other two variables are used to perform calculations, but not returned, as their final values are inessential to the user. In order to achieve this, we utilize the fact, that fixpoints are SBQL queries, and as such may be utilized in other queries. The fixpoint is broken down by utilizing three variables. 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. They will be identified by the optimizer as belonging to different strata, than the engineParts equation. The same principle is used in the next example. The query calculates the total cost and mass of the engine:

```
fixpoint {
  engine : struct{Part,integer} := (Part where name="engine") as x, 1 as howMany;
  engineParts : struct{Part,integer}[1..*] := engine union
    engineParts.Component.((leadsTo.Part) as x,
    (amount*howMany) as howMany);
  detailsMass : float := sum((engineParts where x.kind = "detail").
```

```

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

```

Example 40 Problem decomposition using a fixpoint system

When compared to the transitive closure query presented in Example 21 the fixpoint system shown above is longer (in terms of lines of codes and number of characters), but has the advantage of being easier to write, understand and modify – thanks to breaking down the problem into smaller, easily manageable parts.

2. Workflow

Just like BOM problems, also workflow problems may be easily expressed using fixpoint systems. Our workflow schema example is slightly more complex than the BOM example – we may use it to show more complicated queries.

Let's start – again - with a simple fixpoint system, which finds all activities that may be reached via transitions from a certain activity – e.g. one named "Planning" – this is similar to the transitive closure query presented in Example 16:

```

fixpoint {
activities : Activity[1..*] := (Activity where name="Planning") union
        (outgoing.Transition.leadsTo.Activity);
}

```

Example 41 – Simple fixpoint system over workflow metamodel

The fixpoint system is understandable enough, but let's split the equation into two – one selecting the initial activity, and the other performing the graph traversal. This will be a good starting point for further, more complex queries.

```

fixpoint {
  initial : Activity:= (Activity where name="Planning");
  activities : Activity[1..*]:= initial union (outgoing.Transition.leadsTo.Activity);
}

```

Example 42 – Simple fixpoint system over workflow metamodel

A typical workflow problem is finding the critical path. The first step towards finding the critical path will be calculating the amount of time needed to finish all the tasks on a certain path – each activity will be represented by a pair: activity itself and time required to finish all activities leading to it.

```

fixpoint {
  initial : struct{Activity,integer} := (Activity where name="Planning") as x, x.maxTime as howLong;
  activities :struct{Activity, integer}[1..*]:=initial union
    (activities.x.outgoing.Transition.leadsTo.Activity as x,
    activities.howLong + x.maxTime as howLong);
}

```

Example 43 Simple fixpoint system over workflow metamodel

If we want to, we may use the fixpoint as a part of another query – one that will return the length (in terms of time, not number of tasks) of the critical path – by using the fixpoint system as a parameter of a max function, as shown below:

```

max(fixpoint {
  initial : struct{Activity,integer} := (Activity where name="Planning") as x, x.maxTime as howLong;
  activities :struct{Activity, integer}[1..*]:=initial union
    (activities.x.outgoing.Transition.leadsTo.Activity as x,
    activities.howLong + x.maxTime as howLong);
})
.activities.howLong)

```

Example 44 Use of a fixpoint system as function parameter

The fixpoint system returns two variables (initial and activities) – we need to select the variable that is of interest to us (activities), and then select the element of the structure named howLong – the max function will then operate on a bag of numbers.

Knowing the length of the critical path is usually not enough – we want to know, what activities lie on the critical path. In order to find them, we'll gather the activities on a path as a third element in the equation variable's structure:

```

fixpoint {
initial : struct{Activity,integer, Activity[1..*]} := (Activity where name="Planning") as x,
           x.maxTime as howLong, x as path;
activities :struct{Activity, integer, Activity[1..*]}[1..*]:=initial union
           (activities.x.outgoing.Transition.leadsTo.Activity as x,
           activities.howLong + x.maxTime as howLong, (activities.path union x) as path);
}

```

Example 45 Finding possible workflow execution paths using fixpoint system

Now, we can use this fixpoint in order to select the critical path. In order to do that we'll introduce another two equations – one that will find the time required to finish activities on the critical path (maxTime variable) and another, that selects the path from the activities variable.

```

fixpoint {
initial : struct{Activity,integer, Activity[1..*]} := (Activity where name="Planning") as x,
           x.maxTime as howLong, x as path;
activities :struct{Activity, integer, Activity[1..*]}[1..*]:=initial union
           (activities.x.outgoing.Transition.leadsTo.Activity as x,
           activities.howLong + x.maxTime as howLong, (activities.path union x) as path);
maxTime: integer:=max(activities.howLong);
criticalPath:Activity[1..*]:= (activities where howLong=maxTime).path;
}
.(criticalPath,maxTime)

```

Example 46 Identification of critical path using fixpoint system

We may be also interested in resources required for a certain workflow. The example below shows a fixpoint system, which first traverses the graph of activities in order to find all activities following the "Planning" activity, then finds, what resources are necessary to complete the activities, and finally (using two equations) groups the distinct resources and amounts of those resources needed. Only one equation in this fixpoint system needs to be calculated recursively – the activities equation, but all the other equations serve as a useful

tool for problem decomposition – in the case of a transitive closure query producing equivalent results the query readability would be questionable.

```

fixpoint {
initial : Activity := (Activity where name="Planning");
activities : Activity[1..*] := initial union
    (outgoing.Transition.leadsTo.Activity);
resources : struct{RequiredResource,integer}[1..*]:= activities.requires.RequiredResource as res,
    res.amount as howMany;
distinctResources: RequiredResource[1..*]:= distinct(resources.res);
groupedResources: struct{RequiredResource,integer}[1..*]:= distinctResources as resource,
    sum((resources where res=resource).howMany);
}

```

Example 47 Identification of required resources using fixpoint system

The two previous examples may be easily combined in order to create a fixpoint system that will return all the resources required to complete activities located on the critical path – as shown in the example below.

```

fixpoint {
initial : struct{Activity,integer, Activity[1..*]} := (Activity where name="Planning") as x,
    x.maxTime as howLong, x as path;
activities :struct{Activity, integer, Activity[1..*]}[1..*]:= initial union
    (activities.x.outgoing.Transition.leadsTo.Activity as x,
    activities.howLong + x.maxTime as howLong, (activities.path union x) as path);
maxTime: integer:=max(activities.howLong);
criticalPath:Activity[1..*]:= (activities where howLong=maxTime).path;
resourcesCP : struct{RequiredResource,integer}[1..*]:= criticalPath.requires.RequiredResource as res,
    res.amount as howMany;
distinctResources: RequiredResource[1..*]:= distinct(resourcesCP.res);
groupedResources: struct{RequiredResource,integer}[1..*]:= distinctResources as resource,
    sum((resources where res=resource).howMany);
}

```

Example 48 Identification of critical path resources using fixpoint system

IX. Optimization of Fixpoint Systems

Fixpoint systems, just like the queries using transitive closure operations, need optimization to be useful in practical applications. Similarly to the transitive closure operators, fixpoint system optimization is based on query rewriting. Some of the methods used for fixpoint system optimization are the same methods that are used for other queries, one is specific to fixpoint systems.

1. Fixpoint system-specific optimization method

Fixpoint systems may be optimized not only by use of the standard query rewriting techniques used for other queries. There is another method available, one that allows us to avoid unnecessary evaluation of variables in some steps. It is an adaptation of the stratification concept existing in Datalog to the SBQL.

i. Stratified evaluation

If the fixpoint system consists of more than one fixpoint equation, we may encounter a situation in which not every equation has to be evaluated in every iteration. Such situations may or may not appear depending on the dependencies between the equations i.e. how the equations use each other's variables. The dependencies between equations may be analyzed and the equations divided into groups (stratas) that should be evaluated in sequence – the fixed point for the first group should be reached, then the second group should be evaluated and so on, until the fixed point of the entire fixpoint system is reached. A fixpoint system optimized this way is evaluated as follows:

- A new scope is opened on the environment stack
- Each declared variable is initialized with an empty value appropriate to the type defined for it, the variable is then put on the environment stack
- The following steps are carried out in loop
 - Next group of equations is selected for evaluation. The group is evaluated within a loop:

- Each of the queries in the group is evaluated, the results of the queries stored under unique names on the result stack
 - Result of each query are compared to the corresponding variable stored on the environment stack. The result of this comparison are stored, then the results are assigned as the new value of the variable
 - If all results were equal to the value of corresponding variables on the environment stack, the evaluation of this group is ended and next group is selected for evaluation (if there are any left). Otherwise the next iteration of this loop starts
- The variable values are stored on the result stack, the scope opened by the fixpoint statement on the environment stack is closed.

ii. Stratification algorithm

In order to use the stratified evaluation algorithm, we have to stratify the fixpoint equation system first. The stratification process will identify groups of equations dependent on each other and determine the correct order in which these groups should be evaluated.

For stratification purposes we'll treat the entire fixpoint equation system as a directed graph. The vertices of this graph will represent the equations belonging to the fixpoint system. Edges of the graph will represent the dependencies between equations – if equation e_1 uses a variable defined by equation e_2 , then the graph will contain an edge from vertice representing equation e_1 to vertice representing equation e_2 .

In such a graph, it is possible to identify the equation groups by finding the strongly connected components of the graph. The operation may either be carried out as a part of the query optimization process (and result in query rewriting) or just before evaluation of the fixpoint system. There are several algorithms available, that may be used to identify the strongly connected components of a graph. We suggest using the Kosaraju's [49] or Gabow's [50] algorithm. Both, unlike another well-known algorithm for identification of strongly connected components – the Tarjan's algorithm [51] allow us to find all strongly connected components in a graph - even those not reachable from the starting vertice – in a single pass of the algorithm. Both have linear cost in relation to the size of the input graph (sum of the

number of its' vertices and edges), so the optimization process will be inexpensive in comparison to typical cost of evaluating a fixpoint system.

After identifying the strongly connected components, the optimizer will re-write the query to take advantage of that knowledge - the AST will be modified by introduction of nodes representing the equation groups.

iii. Example

Let's consider a simple fixpoint system:

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

Example 49 Fixpoint system to be stratified

This fixpoint system consists of three equations: one for the *engine* variable, one for the *engineParts* variable and one for the *final* variable.

The *engine* equation does not use any other variable names. Thus, there are no graph edges starting in the node representing this equation.

The *engineParts* equation uses the *engine* variable name. There will be a single graph edge starting in the node representing this equation, it will lead to the node representing the *engine* variable.

The *final* equation uses the *engineParts* variable name. There will be a single graph edge starting in the node representing this equation, it will lead to the node representing the *engineParts* variable. The graph will look as follows:

engine : **struct**{*Part,integer*} :=(**Part where** *name*="engine") **as** *x*, **1 as** *howMany*;



engineParts : **struct**{*Part,integer*}[1..*] := *engine union*

(*engineParts.Component*.(*leadsTo.Part*) **as** *x*,

(*amount*howMany*) **as** *howMany*);



final s : **struct**{*Part,integer*}[1..*] := ((*distinct(engineParts.x)*) **as** *y*).

(*y,sum(engineParts where x=y).howMany*));

Figure 7 - Stratification of a fixpoint system

The optimizer will recognize three strata, that will be evaluated in the following order:

1. *engine*
2. *engineParts*
3. *final*

The first strata (containing the equation for *engine*) will be evaluated twice – once to establish its initial value, second time for the second iteration. At this point no variable in the strata will have different value than in previous iteration, so the evaluation of this strata will be finished. Two evaluations will not be expensive, since the query will be rewritten by the optimizer to factor out the query before the fixpoint equation.

The second strata will be evaluated recursively after the evaluation of the first strata is finished. Then the third strata will be evaluated (again, twice) and the result will be returned.

2. Standard methods used for query optimization

Queries used to determine the value of fixpoint system variables will be – in a typical case – executed not one time, but many times during evaluation of a single fixpoint system. Like every other query, a query defining a fixpoint system equation may contain non-algebraic operators and other expensive subqueries (e.g. procedure or function calls). This means, that optimization of those queries is even more important than optimization of queries used in other contexts. Fortunately, since those queries have the same syntax and semantics as any other SBQL query. As a consequence the same techniques may be used for their optimization.

Our options are not limited to the optimization of the individual fixpoint system equations. The fixpoint system itself may be seen as a non-algebraic operator. This allows us to factor out independent subqueries in front of the fixpoint system.

i. Factoring out subqueries

Like every other query, a query defining a fixpoint system equation may contain non-algebraic operators. The method for factoring out subqueries may be used to optimize a fixpoint system equation just like it would be used in any other context.

Names in the subqueries may be even free in the context of the fixpoint system itself – such situations results in a very interesting development. The factoring out operation closely resembles one of the most efficient optimization methods used in Datalog – the so called Magic Sets. If we abstract from the details related to Datalog evaluation model and its philosophy, the basic idea behind Magic Sets becomes simple:

- identify selections within recursion, which may be evaluated only once (“magic predicates”)
- assign names to those selections, replace selections within recursion with call to the newly assigned names
- evaluate those selections before going into recursion

Factoring out independent subqueries – if we’re capable of factoring out the subquery in front of the fixpoint system - plays exactly the same role in SBA, but is more general – as not only selections, but any calculation independent of the fixpoint system, may be factored out.

By treating the fixpoint system itself as a non-algebraic operator, it is possible to utilize the same optimization techniques as with other SBQL queries.

Let us consider the fixpoint system from Example [37]. The first equation:

```
engine : struct{Part,integer} :=(Part where name="engine") as x, 1 as howMany;
```

contains an expression, in which all names are free in the context of the entire fixpoint system:

```
(Part where name="engine") as x, 1 as howMany
```

This expression can be factored out as an independent subquery, transforming the entire query into (we omit other possible optimizations in this example for the sake of clarity):

```
((Part where name="engine") as x, 1 as howMany) as i1).fixpoint {  
engine : struct{Part,integer} :=i1;  
engineParts : struct{Part,integer}[1..*] := engine union  
    (engineParts.Component.(leadsTo.Part) as x,  
    (amount*howMany) as howMany);  
final s : struct{Part,integer}[1..*] := ((distinct(engineParts.x)) as y).  
    (y,sum(engineParts where x=y).howMany));  
}
```

Example 50 Fixpoint system with factored out subquery

Factoring out this expression allows us to evaluate it only once – thus even without stratified evaluation we improve the evaluation time. With stratified evaluation there will be no significant improvement in this particular case, however – as the expression would be evaluated only twice. Let us consider another example, in which the improvement will be more visible:

```
fixpoint {  
engine : struct{Part,integer} :=(Part where name="engine") as x, 1 as howMany;  
engineParts : struct{Part,integer}[1..*] := engine union  
    (engineParts.Component.(leadsTo.Part) as x,  
    (amount*howMany) as howMany) where howMany>avg(Part.amount);  
final s : struct{Part,integer}[1..*] := ((distinct(engineParts.x)) as y).
```

```
(y,sum(engineParts where x=y).howMany));
}
```

Example 51 Fixpoint system with a subquery evaluated multiple times

In this case, the independent subquery:

```
avg(Part.amount)
```

is a part of an equation, which can be expected to be evaluated multiple times before reaching a fixed point. Evaluating this subquery just once will reduce the evaluation time of the entire fixpoint system – and this time the desired result can not be achieved just with stratified evaluation. If we factor out the independent subquery like this:

```
(avg(Part.amount) as t).fixpoint {
engine : struct{Part,integer} :=(Part where name="engine") as x, 1 as howMany;
engineParts : struct{Part,integer}[1..*] := engine union
    (engineParts.Component.((leadsTo.Part) as x,
    (amount*howMany) as howMany) where howMany>t;
final s : struct{Part,integer}[1..*] := ((distinct(engineParts.x) as y).
    (y,sum(engineParts where x=y).howMany));
}
```

Example 52 Fixpoint system from Example 51 after factoring out a subquery

the subquery will be evaluated just once, before the recursive evaluation starts. Just like in the case of the Datalog's Magic Sets – a subquery is executed once, its result stored under a name and then used in order to speed up the evaluation of a recursive calculation.

ii. Pushing out selection

An important question arises – if factoring out independent subqueries can be successfully used with fixpoint equations, is it possible to use the pushing out selections technique too?

We're looking at fixpoint as a non-algebraic operator, with slightly different syntax. As we know, pushing out selection can be used with an operator, which has the distributivity property. So – the real question we need to answer is: is the fixpoint operator distributive?

As we already know, some of the transitive closure operators are not distributive – e.g. the *leaves unique by* operator. The fixpoint system

```
fixpoint{x1 := q1; x2 := q2;... xm := qm}
```

is semantically equivalent to the following query with the leaves unique by operator:

```
(struct( bag() group as x1, bag() group as x2, ... , bag() group as xm )  
leaves unique by  
struct( q1 group as x1, q2 group as x2, ... , qm group as xm ) )  
}
```

Example 53 – Expressing fixpoint system as a transitive closure

The equivalence stems from the fact that in both cases the evaluation procedure is the same. Thus, the fixpoint operator cannot be treated as a distributive operator, and the technique of pushing out selections is not a viable technique for fixpoint system optimization. The individual queries within a fixpoint, however, may still be optimized using this technique.

3. Synergy between stratification and factoring out independent subqueries

Both stratification and factoring out independent subqueries are powerful optimization tools, that may significantly reduce the time of fixpoint evaluation. If they're used together, additional optimization opportunities may be found.

Normally it is impossible to factor out a subquery that utilizes one of the fixpoint variables – as the subquery will not be independent of the fixpoint. However, if the query is stratified, the variable name may be independent in the context of the particular stratum in which the subquery exists. Let's consider the following example:

```
fixpoint {  
engine : struct{Part,integer} :=(Part where name="engine") as x, 1 as howMany;  
allParts : struct{Part,integer}[1..*] := engine union  
    (allParts.Component.(leadsTo.Part) as x,  
    (amount*howMany) as howMany) ;  
engineParts : struct{Part,integer}[1..*] := engine union  
    (engineParts.Component.(leadsTo.Part) as x,  
    (amount*howMany) as howMany) where howMany<avg(allParts.howMany);  
final s : struct{Part,integer}[1..*] := ((distinct(engineParts.x) as y).  
    (y,sum(engineParts where x=y).howMany));
```

```
}
```

Example 54 Complex fixpoint system

The subquery

avg(*allParts.howMany*)

will utilize variable *allParts*, which has been calculated in a previous stratum. Using only factoring out independent subqueries, we cannot utilize this knowledge as factoring out this subquery will force its evaluation before the fixpoint of this particular variable has been found. If we utilize only stratification, the equation will still be evaluated multiple times – and the subquery with it.

It is possible, however, to treat each stratum of the fixpoint system as a separate fixpoint system for the optimization purposes. The optimizer will identify independent subqueries within each stratum and factor them out in front of this particular stratum, so the entire fixpoint system will be of the following form:

fixpoint {*i*₁.*s*₁; *i*₂.*s*₂; ... *i*_{*n*}.*s*_{*n*}}

where *i*₁...*i*_{*n*} are independent subqueries factored out from strata *s*₁...*s*_{*n*}. The evaluation sequence will be as follows:

*i*₁.*s*₁

*i*₂.*s*₂

...

*i*_{*n*}.*s*_{*n*}

Every stratum in the sequence will be evaluated as a separate fixpoint system (naturally utilizing the results of evaluation of previous strata). After evaluation of the last stratum the results of all evaluations have to be combined and returned as the result of the fixpoint system.

Thus, the fixpoint from Example [54] will be evaluated as follows (other possible optimizations are omitted for the sake of clarity):

- 1) Initialization of all variables

2) stratum 1:

```
engine : struct{Part,integer} := (Part where name="engine") as x, 1 as howMany;
```

3) stratum 2:

```
allParts : struct{Part,integer}[1..*] := engine union  
(allParts.Component.((leadsTo.Part) as x,  
(amount*howMany) as howMany);
```

4) **factored out subquery:**

```
avg(allParts.howMany) as t
```

5) stratum 3:

```
engineParts : struct{Part,integer}[1..*] := engine union  
(engineParts.Component.((leadsTo.Part) as x,  
(amount*howMany) as howMany) where howMany<t;
```

6) stratum 4:

```
final s : struct{Part,integer}[1..*] := ((distinct(engineParts.x) as y).  
(y,sum(engineParts where x=y).howMany));
```

Query **avg**(*allParts.howMany*) will be evaluated only once, so the calculation of average number of necessary parts will not be repeated needlessly multiple times. Without the synergy between stratification and factoring out, the subquery might be only factored out within the *engineParts* equation, providing only limited optimization.

4. Detection of non-recursive equations

Fixpoint systems may be used – as has been shown – to decompose complex tasks and describe them using more readable, simple equations.

One of the consequences of this approach is that fixpoint systems may contain equations which do not require more than one evaluation – they will reach their fixed points in the first step. Yet, by default, they will be evaluated two times – in order to determine, whether their value has changed. This imposes certain overhead on the fixpoint systems, which makes them less efficient than other possible solutions.

It is possible to detect such equations and evaluate them only once. What conditions make an equation a candidate for single evaluation? In order to determine this, we really need to specify, what qualifies an equation for multiple evaluations:

- If an equation references its own results from previous iteration – by using its name – it will of course have to be evaluated recursively
- If an equation references the results of another equation, we have two possibilities:
 - the equation references another equation from the same stratum – in this case it will have to be evaluated recursively, as the results of the referenced equation may change in subsequent iterations
 - the equation references only equations from previously evaluated strata – in this case the information required to evaluate the equation is available in the first evaluation, no recursive evaluation is necessary
 - by definition, it is impossible for an equation to reference equation variable calculated in a stratum further in the evaluation order.

This means, that an equation needs to be evaluated only once if it satisfies the following conditions:

- it does not reference itself
- it does not reference other equations in the same stratum

Let's take a look at the fixpoint system from example 54. The "engine" equation does not reference itself or any other equation, thus may be evaluated once. The "allParts" equation references itself – it has to be evaluated recursively, the same applies to "engineParts" equation. The "finals" equation references only equations that have been already evaluated in the previous strata – qualifying for a single evaluation.

This means, that fixpoint system will appear as follows:

```
fixpoint {  
engine : struct{Part,integer} := (Part where name="engine") as x, 1 as howMany;  
allParts : struct{Part,integer}[1..*] := engine union  
    (allParts.Component.((leadsTo.Part) as x,  
    (amount*howMany) as howMany);
```

```
engineParts : struct{Part,integer}[1..*] := engine union  
    (engineParts.Component.(leadsTo.Part) as x,  
    (amount*howMany) as howMany) where howMany<avg(allParts.howMany);  
final s : struct{Part,integer}[1..*] := ((distinct(engineParts.x)) as y).  
    (y,sum(engineParts where x=y).howMany));  
}
```

Example 55 Fixpoint system with equations not requiring recursive evaluation

The grayed-out equations will be evaluated once, equations in black will be evaluated recursively.

X. 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.

1. Procedures –Syntax and Semantics

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

2. Views

SBQL views, as described in [52] 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.

The definition of a sack is the only obligatory part of the view definition. It describes relation between the objects generated by the view and real objects stored in the database (or other virtual objects). It can be an arbitrary complex functional procedure. It returns a set of entities, called *seeds*, that are used to make up virtual objects. Usually a seed is a binder, but it is not required. The important thing is that a function *nested(seed)* cannot return null. One seed corresponds to one virtual object.

The basic syntax for SBQL view is shown below:

view ::= **create view** viewName { definition }

definition ::= sack *other elements*

sack ::= **virtual objects** seedName { sackdef }

sackdef ::= **return** query

seedName ::= name

Where *query* is any proper SBQL query returning one or more seed entities. The view definition may contain other information, besides the sack definition – e.g. definitions required to make the view into an updatable view etc, but they are of little interest from this paper’s point of view.

It is worth noticing, that unlike recursive procedures/functions, recursive views don’t perform recursive calls to themselves. Instead, they allow us to wrap any query – including a call to a recursive function, a fixpoint or transitive closure – in a form that allows reuse and further manipulation.

XI. Recursive functions and views in SBQL – usage examples

1. Recursive functions

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 the amount required to make this part, is shown below:

```
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 )))  
}
```

Example 56 Simple recursive procedure

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 of procedure call below shows how this procedure may take a collection of parameters:

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

Example 57 Recursive procedure call

The main 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 the example below. 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).

```

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 )
}

```

Example 58 Problem decomposition - simple procedure utilizing a recursive procedure

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 written for this purpose is shown below:

```

procedure subParts(whatPart) {
    return distinct(
        fixpoint {
            parts : Part[1..*] = whatPart union
                (parts.Component.leadsTo.Part);
        })
}

```

Example 59 Problem decomposition - simple procedure utilizing a recursive procedure

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. Thanks to this approach, a fixpoint system (or any other query) can be parametrized and made persistent with no need for additional syntax.

2. Recursive Views

Views, unlike recursive functions, transitive closures or fixpoint systems don't introduce a new recursive processing paradigm to SBQL. Instead, they allow us to wrap queries utilizing other recursive processing paradigms into a view definition. For example:

```
create view EnginePartsDef {
    virtual objects EngineParts {
        return distinct(fixpoint {
            parts : Part[1..*] = (Part where name="engine")
            union (parts.Component.leadsTo.Part);
        }) as b;
    }
    on retrieve do
        return b;
}
```

Example 60 Sample view utilizing a fixpoint system as sack

The view utilizes a fixpoint system as a sack, making it possible to retrieve results of the fixpoint system by using *EngineParts* as name in a query.

Similar to the procedure from example 48, this view allows us to make a fixpoint persistent, without the need to introduce any additional syntax to the fixpoint system itself. There are, however some important differences between those two approaches.

The first difference is the inability to pass parameters to the view. A procedure may receive a number of parameters, and pass them on to the queries in its body. View mechanisms that currently exist in SBQL don't allow that. Introduction of parametrized views is possible, but little research has been done in this direction so far.

The second difference is the potential for results caching. Since views will usually be introduced for often executed queries, introduction of facilities for view results caching may be worthwhile – regardless of whether they'll require database administrator's decision to cache view results or the decision will be automatic. Introduction of similar facilities – especially ones with automated decision making process – for procedure and functions is less desirable, as they will as often be used for simple calculations, perform operations with side effects etc.

XII. Optimization of recursive procedures

Recursive procedures represent a completely different approach to recursive querying than fixpoint systems or queries using the *close by* family of operators. While queries (including fixpoint systems) are an example of declarative programming, recursive procedures (even ones utilizing queries – e.g. procedures in SBQL) are examples of imperative programming.

Declarative programming describes the programmer's intent – data he wishes to retrieve and suggests a way of retrieving the data (the query executed "as provided"). The knowledge of the database's metamodel, available indices etc. may be utilized to re-write query to in such a way its semantics will become unchanged (so the programmer's intent will be fulfilled), but the execution will be faster.

In the case of imperative programming the programmer does not state his intent, but provides an algorithm describing the steps that have to be executed in order to reach his goal. This allows the programmer a much greater freedom of expression, but at the same time it makes the optimization problem a more difficult one. Any optimization performed must be very limited in its scope, in order not to modify the algorithm itself.

In most programming languages, the optimization of code is performed during compilation, using the intermediate code or abstract syntax tree generated from the source code as input to the optimization process. The intermediate code (or AST) is re-written and executable code (or bytecode) is generated from it. Optimization of code in interpreted languages is relatively rare, in most cases the optimizer is really a compiler to some form of bytecode, bundled with a new runtime capable of executing that bytecode. This is a result of the nature of interpreted languages – modification of source code by the optimizer is not desirable, and optimization before every execution of program by modification of the AST generated by the interpreter may turn out to actually result in extending the total time required to execute the program (if the time necessary to execute the optimized program + time necessary to re-write the AST is longer than time necessary to execute unoptimized code).

Compilers perform optimization tasks that may be roughly assigned to two groups – optimization of execution time and reduction of the size of generated executable code. We're mostly interested in optimization of execution time – below we'll discuss method of optimization described in [53].

1. Removal of shared sub-expressions

An expression E is called a shared sub-expression, if E was evaluated earlier and none of E 's variables have changed since its last evaluation. This means, that the technique is very similar to factoring out independent subqueries – an expression is evaluated only once, and the value is used one or more times later. There are, however, important differences.

A query is evaluated within a single transaction, and, unless it uses procedures with side effects affecting the data on which the independent subquery is based, the independent subquery's value will not change – so, after spotting an independent subquery it may be factored out without any further analysis.

In the case of expressions within procedures and functions, however, the problem of identification of shared sub-expression is more complicated – decision, whether an expression (or its part) is a shared sub-expression has can be made only after close examination of the procedure and all procedures called by it between the point where the expression is first evaluated and the point where we want to re-use it.

In order to find expression that may be used as an shared sub-expression the following steps have to be taken for each expression encountered in the procedure:

1. Store the expression as a variable E
2. Analyze each subsequent statement in the procedure (this **has** to be done in the sequence of their appearance in the source code)
 - a. If the statement contains an expression identical to E , E is a shared sub-expression
 - i. Create a local variable V , assign E as its value, replace both occurrences of E with references to V

- b. If the statement modifies one or more variables, database objects etc. that were basis for evaluation of E , discard E – it will not be a shared sub-expression
- c. If the statement is a procedure call perform step 2 (except for 2a) for the invoked procedure as well.

Removal of shared sub-expressions within a procedure may – in author's opinion – result in considerable improvement of execution time, if the shared sub-expression is a query referring to objects in data store and not just expression using local variables. However, such expressions will be usually identified by the programmer and appropriate steps will be taken by him. Optimizer will usually be left to deal just with expressions utilizing local variables.

Utilization of this method will or will not be desirable, depending on the implementation of SBQL – while optimizing them in a compiled variant would be desirable, the optimization process in an interpreted language would in most cases just degrade performance – as it would be more expensive than simply evaluating the expression again.

XIII. Transitive Closures, Fixpoint Equation Systems and Recursive Functions – an Attempt at Comparison

SBQL offers the use of three different paradigms for recursive processing: transitive closure queries, fixpoint equation systems and recursive functions. We've presented all three, it may be worthwhile to compare their relative usefulness.

1. Expressive Power

First let's start with the power of all three paradigms. The relative power of all three is very similar – it is possible to implement both fixpoint equation systems and transitive closures utilizing recursive procedures (or even non-recursive, by replacing recursion with iteration), thanks to the availability of loops, recursive calls, ability to use queries as parameters and in expressions.

Fixpoint equations systems may be expressed using transitive closures, as shown in example 53. Similarly, a transitive closure may be easily expressed as a fixpoint equation.

Procedures and functions, by definition are Turing-complete in terms of expression power. While transitive closure queries and fixpoint systems are based on the concept of declarative programming, they offer the same expression power – as they allow us to use the same control structures as procedures do (every SBQL query allows that) and also may introduce side effects by making changes to the state of the system – both practices are possible, but not recommended. They also may invoke procedures and functions, if desired.

2. Optimization Potential

Optimization potential plays an important role in the usefulness of a recursive processing paradigm. Since recursive queries will usually be expensive, the usefulness of a paradigm that is powerful, but offers little optimization potential, will be often very limited.

As shown in the respective sections, both fixpoint equation systems and transitive closures offer various opportunities for optimization – both optimization by rewriting and by query results caching. Of these two, fixpoint equation systems have more optimization potential, as the synergy between stratification and factoring out independent subqueries may – in some situations – significantly reduce the evaluation time. This advantage, however, may be considered to be nullified by slightly less efficient evaluation algorithm that requires the comparison between results of current and previous iteration in order to decide, whether a fixed point has been reached. Overall, both paradigms may be considered to be equivalent in this area.

Recursive procedures are at serious disadvantage when it comes to optimization. As they are an example of imperative programming, the intent behind a procedure or a set of procedures is unknown to the optimization engine – thus the ability to rewrite procedure code in order to reduce execution time without changing the results is limited. Expressions may be optimized utilizing the same techniques as in the case of other queries (as expressions in SBQL are queries), procedures may also be rewritten using optimization techniques known from programming language interpreters and compilers (like the removal of shared sub-expressions technique described in previous section). It is, however, impossible to automatically recognize and rewrite a sub-optimal algorithm.

3. Familiarity and Ease of Use

The success or failure of many programming and query languages is often decided not by the performance of programs written in them, their elegance or other qualities – but by how approachable they are for a typical programmer. There are many examples proving this statement – flawed, poorly constructed languages, comparing poorly in most areas to other, more elegant languages have gained immense popularity. From BASIC to PHP, languages whose only quality was the ease with which their basics could be learned have succeeded, where languages gaining recognition for their elegance have failed.

Introduction of recursive processing paradigm, which is too difficult to use, may be pointless – as the effort spent on its design and implementation could be better spent on improving the more usable approaches.

Recursive procedures enjoy serious advantage in this area, by the virtue of being approach most programmers are familiar and comfortable with – imperative programming is the first paradigm learned by most professional programmers. It is also easier to introduce side effects using recursive procedures.

The difference between usability of transitive closures and fixpoint equations is a more complex problem. In the case of simple recursive queries, transitive closure will probably be the preferred approach – the code is shorter and simpler, the programmer does not have to provide type declarations – as the result, a simple transitive closure query is easier to write than comparable fixpoint equation system.

In the case of more complex queries, the qualities of the fixpoint systems become more visible. Fixpoint equation system allows easy problem decomposition into multiple equations – which is equivalent to the use of **as** and **group as** operators in transitive closure queries, but more readable. The ability to write a query in stages easily will – in author’s opinion - compensate for limited familiarity with the concept and the need to provide type definitions for every variable.

4. Comparison Conclusions

All three paradigms offer similar expression power – and the ability to combines them reinforces this even further. In other categories the imperative paradigm – use of recursive functions – shows properties exactly opposite to the declarative approach, by being more familiar (and thus probably easier to use), but much more difficult to optimize, than declarative queries.

Choosing a single approach as the “preferred one” is impossible. Instead, the author chooses to provide a recommendation on how and when each paradigm should be used. A strong emphasis should be put on correct utilization of all three querying paradigms, especially during training of programmers learning SBQL or its derivatives.

Recursive procedures should be used mostly in applications, in which side effects are desirable, or in which it is critical that some processing algorithm is followed exactly. Procedures should also be used when it is necessary to make a query persistent and possible

to parametrize (unless facilities for parametrized views are introduced). In the case of queries which do not require parameters, views will be the preferred persistence facility.

Transitive closures should be utilized mostly in ad-hoc queries, that do not require very complex processing. More complex transitive closures will usually be too difficult to easily understand and, as a result, very error-prone.

Fixpoint systems should be utilized mostly for more complex queries that don't require side effects. Usually they will be utilized in conjunction with views or recursive procedures as means for achieving persistence of the query. The easy problem decomposition gives them serious advantage over transitive closures in the case of more complex queries – this should be utilized.

XIV. Implementation of Transitive Closure operators in ODRA DBMS

ODRA DBMS implements four transitive closure operators:

- *close by* operator – the basic transitive closure operator
- *leaves by* operator – transitive closure operator returning only the leaf elements of the result graph
- *close unique by* operator – transitive closure operator, which removes duplicate elements from the result set to avoid cycles
- *leaves unique by* - transitive closure operator returning only the leaf elements of the result graph, which removes duplicate elements from the result set to avoid cycles

All four operators are implemented as SBQL non-algebraic operators.

The full description of architecture of ODRA DBMS may be found in [54]. As this thesis deals with language extensions, the modifications required for implementation are located within the ODRA's query evaluation engine.

The engine's architecture is a pipeline:

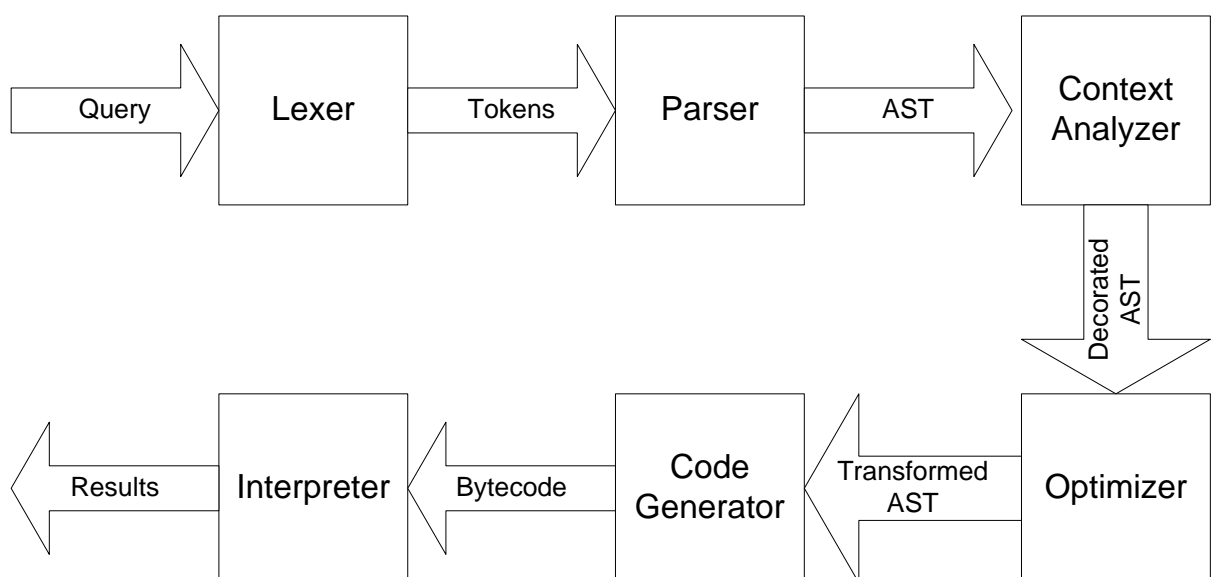


Figure 8 Query evaluation engine - architecture

1. Lexer and Parser

Lexer, built using the JFlex lexer generator is responsible for breaking down SBQL source code into tokens, which are further processed by the parser. Parser is built using the CUP parser generator, and is responsible for generation of abstract syntax tree (AST) of an ad-hoc query or SBQL program module.

The implementation in this module of query evaluation engine introduced new tokens – one for each operator and added them to the language grammar as binary operators. Classes representing the operators in AST were also introduced

2. Context analyzer

Context analyzer is responsible for static evaluation of SBQL ad-hoc queries and programs. The main goal of static evaluation are typechecking, detection of references to undeclared names and identification of sections of AST in which particular names are bound. Context analyzer, as well as Optimizer and Code Generator are based on the Visitor design pattern.

Transitive closure implementation introduced functionality required by the context analyzer to properly handle the new AST classes representing transitive closure operators.

3. Optimizer

Optimizer is responsible for transformation of SBQL queries into semantically equivalent form, in order to reduce the evaluation cost of the query. ODRA's optimizer utilizes query rewriting techniques described in [48], manipulating the AST in order to achieve the desired goals. Utilized techniques include: dead query removal, pushing out selections, factoring out independent subqueries and utilization of indices.

Implementation introduced pushing out selections for the *close by* and *leaves by* operators (as the other two operators are non-distributive, see VII.1.iii) and factoring out independent subqueries for all four operators.

4. **Code generator**

Code generator transforms the SBQL AST into bytecode. The Juliet bytecode represents basic operations of the SBQL virtual machine, the language is described in [54]. The bytecode is then used by the interpreter (see below) to evaluate the query.

Generation of bytecode for all four transitive closure operator has been implemented. The evaluation algorithms described in V.1 were used.

5. **Interpreter**

The interpreter module executes Juliet bytecode in order to evaluate the required ad-hoc query or SBQL program. In order to do so, it utilizes information provided by other modules of ODRA DBMS: Persistent and Transient Object Managers, Module Manager, Result Decoder (in the case of distributed queries) and of course the Code Generator. Results are encoded and sent to the client using the Result Encoder module.

As the interpreter utilizes the standard set of operations of Juliet bytecode, and all four transitive closure operators are translated to the bytecode, no implementation changes were necessary in this module.

XV. Implementation of Fixpoint Systems in ODRA DBMS

Fixpoint systems implementation is a part of the query evaluation engine described in XV. Fixpoint systems within this framework are implemented as follows:

1. Lexer and Parser

Token for fixpoint system has been introduced into the lexer. Fixpoint system has been introduced into grammar as a type of expression – which allows the user to formulate more complex queries using fixpoint systems as subqueries.

Classes representing the fixpoint equation system and its elements in Abstract Syntax Tree were introduced. Also, a class representing a fixpoint equation system's stratum has been implemented.

The object representing the fixpoint equation system stratifies the equation system using Kosaraju's [49] algorithm after its initialization. The stratification may be performed at this point, as the syntax tree is constructed bottom-up, so all necessary information is available, and it has to be performed before the AST is decorated by the context analyzer in order to take advantage of optimization method described in Chapter X.3. In future, the stratification process may be refactored in order to move it to the context analyzer module.

2. Context analyzer

Functionalities required by the context analyzer to properly typecheck the fixpoint systems and expressions containing fixpoint systems, as well as those required to properly decorate AST for optimization purposes have been implemented.

3. Optimizer

Implementation enabled the optimizer to traverse the equations of a fixpoint system in order to perform local optimizations by pushing out selections and factoring out independent subqueries.

4. Code generator

Generation of bytecode for fixpoint systems has been implemented. In order to take advantage of the optimization methods described in Chapter X, the generated code evaluates the fixpoint system according to a modified algorithm:

- A new scope is opened on the environment stack
- Each declared variable is initialized with an empty value appropriate to the type defined for it, the variable is then put on the environment stack
- The following steps are carried out in loop:
 - Queries factored out (according to the rules in Chapter X.3) and detected non-recursive equations (Chapter X.4) are evaluated (if any)
 - The evaluation of next stratum begins (if one exists)
 - Each of the queries in the stratum is evaluated, the results of the queries stored under unique names on the result stack
 - Result of each query are compared to the corresponding variable stored on the environment stack. The result of this comparison are stored, then the results are assigned as the new value of the variable
 - If all results were equal to the value of corresponding variables on the environment stack, the loop is ended. Otherwise the next iteration of this loop starts
- The variable values (from environment stack) are stored on the result stack, the scope opened by the fixpoint statement on the environment stack is closed.

5. Interpreter

Similarly to the case of transitive closures, no implementation changes to the interpreter were necessary.

XVI. Further research

Recursive queries are an area, which offers many opportunities for further research.

1. Datalog-like rules

Current approach to fixpoint equation systems in SBA is quite different to the one chosen in Datalog – transient sets of fixpoint equations, instead of sets of rules declared separately as persistent entities in the database. Research into datalog-like rules for SBQL may result in introduction of new, possibly useful, facilities into the language. While, in author's opinion, explicit declaration of all rules required to calculate the result of a query will be more useful in most situations, the Datalog approach also has some merits – especially, when combined with a powerful query language like SBQL. There are several possible approaches to implementation of such rules – e.g. they may be implemented using recursive functions, parametrized views, by introducing new entities to the data model. The research would involve finding the best method of implementing the rules, research of optimization techniques and investigation into the best ways to utilize them.

2. Recursive queries in distributed databases

Optimization of recursive queries is especially important in distributed (or federative) databases – as evaluation of query may result in heavy traffic between database servers. Some research and development work on SBA in general and the ODRA DMBS in particular has been focused on implementation of distributed processing facilities in the system – including distributed query optimization. Evaluation of a recursive query may result in multiple requests for data to the same database server –generating heavy network load and seriously increasing the time required for query evaluation due to network communication overheads.

Development of optimization techniques for distributed recursive processing is vital for practical applications of SBQL. The optimization techniques may be both adaptations of existing optimization techniques for non-recursive queries, as well as heuristics utilizing

knowledge about data distribution between servers in question and performance of past queries.

3. Alternative storage models

Recently some relational database management systems (e.g. MySQL) have introduced alternative storage models, which considerably improve their performance in specific applications. Research into possible alternative organization of data in memory and on disk may result in performance improvement in recursive query evaluation.

Column-oriented storage, described in e.g. [55], as well as Correlation storage described in e.g. [56] greatly increase the performance of some classes of queries. Investigation into alternative storage model for the ODRA DBMS – one aimed at performance improvement for recursive queries may prove worthwhile, especially if the research were to be combined with research into datalog-like rules, recursive query optimization and general optimization techniques.

4. Parametrized views

Currently the views mechanism, despite being much more powerful than one available in SQL and other query languages, has one drawback – it does not provide facilities to define views accepting parameters. Parametrized views may be useful as a way of declaring persistent, parametrized recursive queries – as well as one of potential ways of declaring datalog-like deductive rules. The research would have to cover both syntactical and semantical aspects of parametrized views, as well as usability problems.

XVII. Conclusions

In this thesis we've shown, that recursive queries may be relatively simple to introduce and implement in efficient manner, if the theoretical foundations of a query language are flexible and sound. In such case, both syntax and semantics of recursive queries may be clean, simple and easy to understand – making them a practical element of the language, not just a curiosity.

We've shown, how optimization techniques known from Stack Based Approach and from other query languages (e.g. Datalog) as well as programming languages can be used and combined in order to improve the performance of database management system processing such queries.

We've also provided examples of practical applications for each of three recursive querying paradigms, discussing the construction of those queries – an part of this thesis, which may be useful as an educational tool, as often the easiest way to grasp a concept is to learn from examples.

Finally, we've discussed the relative merits of all three paradigms as they appear in SBQL and presented the author's opinion on preferred utilization of each of those three paradigms, based on criteria of expression power, expected performance based on optimization techniques that may be utilized for each paradigm as well as their ease of use and familiarity of programmers with each of these concepts.

XVIII. List of Figures

Figure 1 - simple database schema	18
Figure 2 - Sample database schema	48
Figure 3 - Example of SBA data store	49
Figure 4 - Example of ES states during evaluation of a query.....	49
Figure 5 - BOM schema	65
Figure 6 – Simple workflow metamodel	69
Figure 7 - Stratification of a fixpoint system	98
Figure 8 Query evaluation engine - architecture	121

XIX. List of Examples

Example 1 Transitive closure query in Oracle	18
Example 2 Transitive closure with result formatting	20
Example 3 Transitive closure query in DB2	24
Example 4 Transitive closure In DB2 – with result formatting.....	25
Example 5 Datalog fact.....	30
Example 6 Datalog rule	30
Example 7 Datalog- rule	31
Example 8 Datalog- program with two minima models.....	32
Example 9 Recursive procedure in PL/SQL.....	38
Example 10 - Function Definition in XQuery	40
Example 11 - Function call for Example 10	40
Example 12 - SBQL function	52
Example 13 - SBQL function call.....	52
Example 14 Square Root calculation.....	62
Example 15 Square Root calculation with a stop condition	63
Example 16 Simple transitive closure over BOM schema.....	65
Example 17 Transitive closure over BOM schema.....	66
Example 18 Complex transitive closure over BOM schema	66
Example 19 Complex transitive closure over BOM schema	67
Example 20 Complex transitive closure over BOM schema	67
Example 21 Complex transitive closure over BOM schema	68
Example 22 – Simple transitive closure query over workflow metamodel	69
Example 23 Transitive closure query over workflow metamodel	70
Example 24 Transitive closure query over workflow metamodel	70
Example 25 Transitive closure query over workflow metamodel	71
Example 26 Complex transitive closure query over workflow metamodel.....	71
Example 27 Complex transitive closure query over workflow metamodel.....	72
Example 28 – Transitive query with an independent subquery – before factoring out.....	75
Example 29 – Transitive query with an independent subquery – after factoring out.....	76

Example 30 Query with a predicate independent of the close by operator	77
Example 31 The same query after optimization	78
Example 32 Simple fixpoint system.....	84
Example 33 fixpoint equation for the purposes of type inference	84
Example 34 Fixpoint system with three equations	85
Example 35 Stratified fixpoint system.....	85
Example 36 Fixpoint system with two equations in the same stratum.....	85
Example 37 Fixpoint system with infinitely nesting binders.....	86
Example 38 Simple fixpoint system with a single equation	89
Example 39 Fixpoint system over the BOM schema	90
Example 40 Problem decomposition using a fixpoint system	91
Example 41 – Simple fixpoint system over workflow metamodel.....	91
Example 42 – Simple fixpoint system over workflow metamodel.....	92
Example 43 Simple fixpoint system over workflow metamodel.....	92
Example 44 Use of a fixpoint system as function parameter	92
Example 45 Finding possible workflow execution paths using fixpoint system	93
Example 46 Identification of critical path using fixpoint system	93
Example 47 Identification of required resources using fixpoint system.....	94
Example 48 Identification of critical path resources using fixpoint system	94
Example 49 Fixpoint system to be stratified	97
Example 50 Fixpoint system with factored out subquery	100
Example 51 Fixpoint system with a subquery evaluated multiple times.....	101
Example 52 Fixpoint system from Example 51 after factoring out a subquery.....	101
Example 53 – Expressing fixpoint system as a transitive closure	102
Example 54 Complex fixpoint system	103
Example 55 Fixpoint system with equations not requiring recursive evaluation.....	106
Example 56 Simple recursive procedure.....	110
Example 57 Recursive procedure call	110
Example 58 Problem decomposition - simple procedure utilizing a recursive procedure	111
Example 59 Problem decomposition - simple procedure utilizing a recursive procedure	112
Example 60 Sample view utilizing a fixpoint system as sack	112

XX. Bibliography

1. Khalaila, A., Eliassen, F.: An efficient bill-of-materials algorithm, Tekniske rapporter / Institutt for informatikk 31, pages., University of Tromsø, Tromsø (1997)
2. Afrati, F., Gergatsoulis, M., T., K.: Answering Queries Using Materialized Views with Disjunctions. In : Lecture Notes in Computer Science 1540, pages 435-452. Springer, Berlin (1999)
3. Fotouhi, F., Johnson, A., Rana, S., Rakesh: A hash-based approach for computing the transitive closure of database relations. In : The Computer Journal vol. 35 no. 3, pages 251-259. Oxford University Press, Oxford (1992)
4. Dong, G., Libkin, L., Su, J., Wong, L.: Maintaining transitive closure of graphs in SQL. In : International Journal of Information Technology, pages 46-78. Singapore Computer Society, Singapore (1999)
5. Z. Li, K.: On the Cost of Transitive Closures in Relational Databases. In : Columbia University Technical Report CUCS-004-93. Columbia University, New York (1993)
6. W3C: XML Query Use Cases. Available at: <http://www.w3.org/TR/xquery-use-cases/>
7. Oracle: PL/SQL User's Guide and Reference. Available at: http://download.oracle.com/docs/cd/B10501_01/appdev.920/a96624/toc.htm
8. IBM: Examples of recursive common table expressions. In: DB2 Version 9.1 for z/OS Solutions Information Center. Available at: http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/index.jsp?topic=/com.ibm.db2.9.doc.apsg/db2z_createcte.htm
9. Microsoft Corporation: Recursive Queries Using Common Table Expressions. In: SQL Server Developer Center. Available at: [http://msdn.microsoft.com/en-us/library/ms186243\(SQL.90\).aspx](http://msdn.microsoft.com/en-us/library/ms186243(SQL.90).aspx)

10. Abiteboul, S., Hull, R., Vianu, V.: Part D: Datalog and Recursion, pages 271-465. In : Foundations of Databases. Addison-Wesley (1995)
11. Atkinson, M., Buneman, P.: Types and Persistence in Database Programming Languages. In : ACM Computing Surveys Volume 19 , Issue 2, pages 4-15 19. ACM, New York, NY (1987)
12. Yan, W., N.M., M.: Transitive Closure and the LOGA + - Strategy for its Efficient Evaluation. In : Lecture Notes in Computer Science 364: Mathematical Fundamentals of Database Systems, pages 415-428. Springer, Berlin (1989)
13. S.Taylor, N. I. H.: A Direct Algorithm for Computing the Transitive Closure of a Two-Dimensionally Structured File. In : Lecture Notes in Computer Science 495. 1991, Berlin (Springer)
14. Ullman, J. D., Widom, J.: Chapter 5: Algebraic and Logical Query Languages. In : A First Course in Database Systems. Prentice Hall (1997)
15. Dobrovnik, M., T., M.: Architectural Considerations for Extending a Relational DBMS with Deductive Capabilities. In : Proceedings of SPSE 92, pages 82-93. Springer, Berlin (1992)
16. Wong, L.: Incremental recomputation of recursive queries with nested sets and aggregate functions. In : LNCS 1369: Proceedings of 6th International Workshop on Database Programming Languages. Springer (1997)
17. G. Dong, L.: Properties of Languages That Make Recursive Views Unmaintainable. Available at: <http://citeseer.ist.psu.edu/4739.html>
18. Gallaire, H., Minker, J., Nicolas, J.: Logic and Databases: A Deductive Approach. In : ACM Computing Surveys Volume 16 , Issue 2, pages 153-185. ACM, New York, NY (1984)
19. Ceri, S., Gottlob, G., Tanca, L.: What You Always Wanted to Know About Datalog (And Never Dared to Ask). In : IEEE Transactions on Knowledge and Data Engineering Volume 1 , Issue 1, pages 146-166. IEEE Educational Activities Department, Piscataway, NJ

(1989)

20. P.R. Falcone Sampaio, N. W.: Deductive Object-Oriented Database Systems: A Survey. In : LNCS: Proceedings of the Third International Workshop on Rules in Database Systems. Springer (1997)
21. Liu, M.: OLOG: A Deductive Object Database Language. In : Proceedings of the Workshop on Next Generation Information Technologies and Systems, pages 120-137. Springer, Berlin (1999)
22. Li, X., Liu, M.: Design and Implementation of the OLOG Deductive Object-Oriented Database Management System. In : Lecture Notes In Computer Science 1873: Proceedings of the 11th International Conference on Database and Expert Systems Applications, pages 764-773. Springer, Berlin (2000)
23. Sampaio, P., W., P.: Deductive Queries in ODMG Databases: the DOQL Approach. In : Proceedings of the 5th International Conference on Object-Oriented Information Systems, pages 57-74. Springer, Berlin (1998)
24. W3C: XQuery 1.0: An XML Query Language. Available at: <http://www.w3.org/TR/xquery/>
25. Immerman, N.: Relational Queries Computable in Polynomial Time. In : ACM Symposium on Theory of Computing, pages 147-152. ACM, San Francisco (1982)
26. Freire, J.: Practical Problems in Coupling Deductive Engines with Relational Databases. In : Proceedings of the 5th International Workshop on Knowledge Representation Meets Databases (KRDB '98): Innovative Application Programming and Query Interfaces, pages 11.1-11.7. CEUR-WS.org, Seattle (1998)
27. Han, J., Lakshmanan, L.: Evaluation of Regular Nonlinear Recursion by Deductive Database Techniques. In : Information Systems Volume 20, pages 419-441. Elsevier Science, Oxford (1995)

28. Kemp, D., Srivastava, D., J., S.: Bottom-up Evaluation and Query Optimization of Well-Founded Models. In : Theoretical Computer Science Volume 146 , Issue 1-2, pages 145-184. Elsevier Science Publishers, Essex (1995)
29. Lu, J., Chen, L., Sycara, K., Lu, J.: Recursive Query Rewriting by Transforming Logic Program. In : International Workshops on Logic-based Program Synthesis and Transformation., Manchester (1998)
30. S. Abiteboul, M.: Fixpoint Logics, Relational Machines and Computational Complexity. In : Journal of ACM. ACM (1997)
31. Duschka, O., R., G.: Answering Recursive Queries Using Views. In : Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Conference on Principles of Database Systems pages 109-116. ACM (1997)
32. Eder, J.: General Transitive Closures and Aggregate Functions. In : Database and Expert Systems Applications, Proceedings of the International Conference in Berlin, pages 54-59. Springer, Berlin (1991)
33. Biswadeep, N.: Implementing Generalized Transitive Closure in the Paradise Geographical Information System. In : Technical Report #1272. University of Wisconsin-Madison, Madison (1995)
34. G. Z. Qadah, L.: Efficient Algorithms for the Instantiated Transitive Closure Queries. In : IEEE Transactions on Software Engineering archive , Volume 17, Issue 3. (1991)
35. R. Agrawal, S.: Direct Transitive Closure Algorithms: Design and Performance Evaluation. In : ACM Transactions on Database Systems Volume 15 Issue 3. ACM (1990)
36. He, H.: Master Thesis: Implementation of Nested Relations in a Database Programming Language. McGill University, Montreal (1997)
37. Hao, B.: Master Thesis: Implementation of The Nested Relational Algebra in Java. McGill University, Montreal (1995)

38. Subieta, K.: Teoria i konstrukcja obiektowych języków zapytań. PJWSTK, Warsaw (2004)
39. Subieta, K.: Semantics of Query Languages for Network Databases. In : ACM Transactions on Database Systems Volume 10 Issue 3, pages 347-394. ACM, New York (1985)
40. Subieta, K.: Denotational Semantics of Query Languages. In : Information Systems Volume 12 Issue 1, pages 69-82. Elsevier Science, Oxford (1987)
41. Subieta, K., Beerl, C., Matthes, F., J.W., S.: A Stack-Based Approach to Query Languages. In : Springer Workshops in Computing: Proceedings of the East-West Database Workshop 1994, pages 159-180. Springer, Berlin (1995)
42. Subieta, K., Kambayashi, Y., J., L.: Procedures in Object-Oriented Query Languages. In : Proceedings of the 21th International Conference on Very Large Data Bases, pages 182-193. Morgan Kaufmann, San Francisco (1995)
43. Subieta, K., Missala, M., K., A.: Report 695: The LOQIS System. Description and Programmer Manual. Institute of Computer Science, Polish Academy of Sciences, Warsaw (1990)
44. Subieta, K.: LOQIS: The Object-Oriented Database Programming System. In : Lecture Notes in Computer Science 504: Proceedings of the 1st International East/West Database Workshop on Next Generation Information System Technology, pages 403-421. Springer, Berlin (1991)
45. Kozankiewicz, H., Leszczyłowski, J., Subieta, K.: Report 950: Updateable Object Views., Polish Academy of Sciences, Warsaw (2002)
46. Stencel, K.: Półmocna kontrola typów w językach programowania baz danych. PJWSTK, Warsaw (2006)
47. Momotko, M.: Ph.D. Thesis: Tools for Monitoring Workflow Processes to Support Dynamic Workflow Changes. Institute of Computer Science, Polish Academy of Sciences,

Warsaw (2005)

48. Płodzień, J.: Ph.D. Thesis: Optimization Methods in Object Query Languages., Warszawa (2000)
49. Cormen, T., Leiserson, C. E., Rivest, R. L., Stein, C.: Introduction to Algorithms, 2nd edition. MIT Press (2001)
50. Sedgewick, R.: Algorithms in C, Third Edition. Addison-Wesley (2002)
51. Robert, T.: Depth-first search and linear graph algorithms. SIAM Journal on Computing Vol. 1 (2) (1972)
52. Kozankiewicz, H.: PhD Thesis: Updatable Object Views. IPIAN, Warsaw (2004)
53. Aho, A. V., R., S., D., U.: Chapter 10: Code Optimization. In : Compilers Principles, Techniques and Tools, pages 554-682 2nd edn. Addison Wesley (2006)
54. Lentner, M.: PhD Thesis: Integracja danych i aplikacji przy użyciu wirtualnych repozytoriów. PJWSTK, Warszawa (2008)
55. Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., Zdonik, S.: C-Store: A column-oriented DBMS. In : Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005, pages 553-564. Springer, Berlin (2005)
56. Powell, J.: Illuminate's Correlation Database Accelerates, Expands BI Queries. In : Enterprise Systems Journal 9 April 2008.
57. Aho, A., Ullman, J.: The Universality of Data Retrieval Languages. In : Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, pages 110-119. ACM, San Antonio, Texas (1979)
58. Bancilhon, F.: On the completeness of query languages for relational data bases. In : Lecture Notes in Computer Science 64, pages 112-123. Springer, Berlin (1978)

59. Lyons, R.: Universal Turing Machine in XSLT. In: B2B Integration Solutions from Unidex. Available at: <http://www.unidex.com/turing/utm.htm>
60. Kozankiewicz, H., Leszczyłowski, J., Subieta, K.: Mediators through Virtual Updateable Views. In : Proceedings of the 5th International Workshop on Engineering Federated Information Systems, pages 52-62. IOS Press, Coventry (2003)
61. Kozankiewicz, H., Leszczyłowski, J., Subieta, K.: SBQL Views - Prototype of Updateable Views. In : Proceedings of 8th East-European Conference on Advances in Databases and Information Systems. Local proceedings, Budapest (2004)
62. Kozankiewicz, H., Leszczyłowski, J., Subieta, K.: Updateable XML Views. In : Lecture Notes in Computer Science 2798: Proceedings of Advances in Databases and Information Systems, pages 381-399. Springer, Berlin (2003)