Warsaw University
Faculty of Mathematics, Informatics and Mechanics

Marta Jadwiga Burzańska

New Query Rewriting Methods for Structured and Semi-Structured Databases

PhD dissertation

Supervisor
dr hab. Krzysztof Stencel

Institute of Informatics
Warsaw University

March 2011

Author's declaration:
aware of legal responsibility I hereby declare that I have written this dissertation myself and all the contents of the dissertation have been obtained by legal means.

....
date *Author's signature*

Supervisor's declaration:
the dissertation is ready to be reviewed

....
date *Supervisor's signature*

Abstract

The following thesis presents novel optimization methods for structured and semi-structured databases. Developed methods are based on query rewriting and focus on reduction of various resources consumption during query execution. The first method achieves this goal by rewriting the initial query execution plan into a new plan based on the *reduce* function. The second method utilizes distributivity property of the navigation operator to reduce the sizes of created intermediate structures. Both methods have been developed for semi-structured databases and works with non-recursive operators. This thesis also presents two algorithms for optimization of recursive queries. One of them was developed for SQL and focuses on optimization of recursive Common Table Expressions by the means of predicate pushing. The other algorithm was developed to optimize the composition of SBQL's transitive closures with aggregation.

This paper also presents the results of experimental tests performed for each of the new methods. The results confirm the effectiveness of those methods in reducing the resource consumption and/or increasing the speed of execution.

Keywords: query rewriting, recursive queries, structured and semi-structured query languages, intermediate structures reduction

ACM Computing Classification System: H.2.4. (Query Processing)

Table of Contents

1 Introduction

Query languages are a primary mechanism of communication between an application and a database. Their construction is based on an assumption that the programmer instead of saying how data is to be searched, should simply focus on what data is actually needed from a database. Algorithms used during query execution are hidden from the programmer, what reduces the time needed to write a query and allows the Database Management System (DBMS) to choose the best algorithm depending on the data size and distribution. The burden of improving database access is the task of the optimizer module, which selects the appropriate query execution plan.

Query optimization is therefore a very important task for a DBMS. For years it has been, and still will be, a subject of research for many scientists. Initially, optimization was limited to application of query transformation rules. Later the rule-based optimization became extended with mechanisms of cost-based optimization that helps the optimizer to choose the least costly execution plan. Nowadays there are many optimization techniques that work at the different stages of optimization: from semantic query rewriting, execution plan transformation rules to cost-based optimization of chosen subset of execution plans.

The following thesis presents a group of new query rewriting algorithms developed for structured (relational and object-oriented) and semi-structured data. The main idea behind those algorithms is to reduce the resource consumption while increasing the speed of query execution. Two of the developed techniques concentrate around optimization of recursive queries. Such queries inherently consume a lot of system resources during evaluation. The first technique has been developed for SQL. It is based on the well-known method of pushing predicates, however it has been studied for novel applications in a new context. The second method optimizes composition of an aggregation function with the operator of the transitive closure. This algorithm is presented using semi-structured query language SBQL as an example.

SBQL was also the basis for the two further algorithms. They deal with composition of two functions or operators from semi-structured query languages. The first algorithm is based on translation of initial query execution plans for language's operators into an execution plan based on the reduce function. The idea behind this translation originated from optimization techniques used in functional programming languages. The second algorithm concentrates on reduction of the size of intermediate structures using distributivity property of navigation operator and aggregation.

The thesis is organized as follows. The first chapter of the following thesis browses through the subject of querying object-relational and semi-structured data. It mentions the most popular data models and query languages used in both cases. It also introduces the concept of recursive SQL queries. The second chapter introduces the concepts of the Stack Based Approach (SBA) and the Stack Based Query Language (SBQL). Those concepts may not be familiar to all of the readers of this thesis, thus they will be discussed in detail. This chapter also introduces an experimental implementation of SBQL based on the Python language. This platform, called the PySBQL, was developed by the author of this thesis as a platform for research, development and testing of optimization algorithms for SBQL.

The third chapter presents an overview of a selection of optimization techniques that are based on query rewriting. For each of the selected techniques its general description is given and some of its variants are presented. This chapter also presents the optimization technique developed for functional languages – the shortcut fusion – which was the inspiration for the newly developed optimization methods, which are the main topic of this thesis.

The fourth chapter discusses two novel techniques developed for compositions of non-recursive operators and built-in functions of the semi-structured query language SBQL. The fifth chapter presents new optimization techniques for recursive queries. For each of the four techniques presented in chapters four and five, a summary of their performance tests is given.

The final chapter concludes and points the further possibilities of research on the subject of reduction of the sizes of intermediate structures.

1.1 Querying Object-Relational Data

The most popular data model is the relational model created by E. F. Codd first published in [Codd70]. In the 1972's paper [Codd72] Codd presented a detailed description of the model and presented two formal models for querying data. In 1995 C. Date and H. Darwen in [Date95] described how the relational model can be used to support object-oriented features of database applications.

The most widely-used database query language is the well-known SQL which originated from Codd's querying data models. This language is under constant development to match the needs of programmers and database users. Current SQL standard includes handling of semi-structured data through XML storage and XML-related features. Starting from the SQL:99 standard, the SQL language has been equipped with recursive queries. This type of queries is called the Recursive Common Table Expressions (RCTE). Nowadays there is some research on optimization of recursive SQL queries, however this topic is still open for new methods and optimization algorithms. The following thesis present a novel optimization technique for this type of queries, therefore they should be presented here in more detail.

Each RCTE query starts with the WITH keyword optionally followed by the RECURSIVE keyword and a header of the Common Table Expression. Such query may be divided into three parts: an initial query also known as the *seed query*, a recursive subquery and an outer query that consumes all the rows generated by recursive computation. The basic syntax structure of a recursive CTE is:

```
WITH RECURSIVE R0 (A01 , . . . , A0n ) AS
     <R0's initial query  UNION ALL  R0's recursive query >
[, [RECURSIVE] R1 (A11 , ..., A1n ) AS
     <R1's initial query  UNION ALL  R1's recursive query >
... ]
< query using R0,R1,R2,... >,
```

Each < R_i definition> must consist of at least one initial SELECT query and at least one recursive SELECT query that contains a call to R_i in its FROM clause. Both initial and recursive subquery may be formed out of a union of more than one SELECT queries. SQL-99 standard also allows for the use of one of the special clauses included into CTE definition after the last recursive subquery. Those special clauses include SEARCH and CYCLE clauses. They are used to put additional limitations on the recursive queries to prevent infinite loops. Other clauses are DEPTH FIRST and BREADTH FIRST used to specify the search order. Example 1.1 presents a recursive common table expression that gathers information on courses' requirements and returns a list of courses that should be completed before attending 'Java_1' course.

Example 1.1 Query calculating required courses for 'Java_1' course.

```
WITH RECURSIVE RequiredCourses (BaseCourse, NeededCId) AS (
  SELECT r.CourseId, r.Requires
     FROM Requirements r
  UNION ALL
  SELECT rc.BaseCourse , r.Requires
     FROM Requirements r, RequiredCourses rc
     WHERE  rc.NeededCId = r.CourseId
     AND  r.Requires IS NOT NULL )
SELECT DISTINCT NeededCId FROM RequiredCourses
WHERE BaseCourse = 'Java_1';
```

The biggest disadvantage of this construction is the lack of efficient algorithms for cycle detection. In some cases the recursive query using CTE may not stop because, according to the specification, the computation stops when a fix-point is reached (no new rows are generated using CTE's recursive subquery). A study of current implementations of recursive common table expressions in the most popular database management systems may be found in [Przy10]

The research on optimization of SQL language has been influenced by research papers on another language used to query relational data. This language is Datalog, which became an area of database researchers' interest around 1977's due to the workshop on logic and databases organized

by H. Gallaire and J. Minker [Gall78]. The research on Datalog was concentrated mostly between the mid-80's and the mid-90's. Many existing optimization techniques for relational databases have been initially developed for this language. Datalog was also a platform for research on optimization of recursive queries.

The construction and evaluation of Datalog falls outside of the scope of this thesis. However, detailed information on the subject may be found in [Viei87, Ceri89, Abit95]. For detailed information on Datalog¬ (Datalog with negation) and stratified Datalog semantics see [Przy86, Apt86, Ullm88]

With the development of the object-oriented programming developers and researchers started to look for an alternative to SQL language for a better support for complex data such as graph data, multimedia. In the 1991 Object Data Management Group (ODMG) started working on a new query language, called the Object Query Language (OQL). First version of ODMG standard was created in 1993, second version in 1997. OQL was modeled after SQL [Catt96] and was planned to become the standard for object-oriented databases. However, the ODMG project was closed in 2001 without a completed specification. There are voices that this specification is not implementable [Subi96].

Parallel to the research on OQL of the ODMG group, in the 1990's Kazimierz Subieta started working on the Stack Based Approach (SBA) and on the Stack Based Query Language (SBQL). Both will be described in the Chapter 2 of this thesis.

1.2 Querying Semi-Structured Data

Semi-structured data models are meant to describe unstructured information, in particular irregular data, but they might as well be used to store structured data. There are many methods of representing semi-structured data. Their main idea is to represent data in some form of labeled, directed graphs or a set notation of tagged label-value pairs. The term semi-structured data was introduced by Shoens et al. in 1993 in a system called Rufus [Shoe93]. Modern semi-structured data model usually combine the ideas of the relational model and the object data model [Catt96]. Among the popular semi-structured models are XML and OEM (for more information on this model see [Papa95]).

In the mid 90s, when the World Wide Web gained popularity, so did the Extensible Markup

Language (XML) [XML1.0, XML1.1]. The main focus of XML was to store documents, however its tree model proved to be efficient for representing semi-structured data and hierarchical data. It is also flexible enough to handle rapidly changing structures and sparse properties. Also, XML is platform-independent and in contrast to HTML, XML separates the logical structure of a document from its layout. This is why XML has quickly become the preferred format for representing and exchanging data on the Web. Example 1.2 presents a small XML document.

Example 1.2. XML document containing information on an employee called John Doe

```
<?xml version="1.0" encoding="UTF-8" ?>
<employee>
     <img src="john_doe.jpg" alt='John Doe'/>
     <name>John Doe</name>
     <address>
          <street>
               <street_num>50h</street_num>
               East Road
          </street>
          <city>Smallville</city>
     </address>
</employee>
```

One of the methods for processing XML documents is to use declarative transformation languages such as XPath [XPath1, XPath2], XQuery [XQuery] and XSLT [XSLT] which are standardized by the W3C consortium.

XML Path language (XPath) is a language based on path expressions that allows the selection of parts of a given XML document. An XPath expressions use series of steps to navigate through XML tree by selecting nodes that satisfy certain properties. The evaluation of an XPath expression returns either a sequence of atomic nodes or a sequence of nodes with their subtrees. XPath also allows some minor computations resulting in values such as strings, numbers or Boolean

values. The popularity of XPath and the fact that it is a subset of XSLT and XQuery resulted in a number of papers devoted to improving XPath's evaluation algorithms. One of such papers [Pary09] presents an optimization technique that works in linear time and has linear complexity in the number of bytes of the processed XML document. In this paper Parys proposed an algorithm for efficient retrieval of nodes satisfying a vast range of XPath queries. Other papers on similar subject are [Gott05, Bene08, Gotz09].

Nowadays the most popular language designed for querying semi-structured data is XQuery [XQuery]. The goal of the design was to provide the expressive power of a query language like SQL and, in addition, to support XML-specific operations such as navigation in hierarchical data. Most features have been influenced by the functionality of Quilt and SQL languages. Other influences come from semi-structured languages like Lorel [Abit97] and XML-QL [XMLQL]. XQuery is a superset of XPath and as such supports richer operations like joins, projections, aggregations, but also supports functionality of a programming language. Nowadays all the major database vendors implement either some subset of XQuery or the full range of features. The most popular XQuery implementations are XQRL/BEA [Flor04], Saxon [Saxon], Sedna [Fomi06], MonetDB [Bonc06], DB2 [Ozca08], Oracle [Liu08], Zorba [Bamf09].

In the recent years the subject of processing XML and XQuery was one of the most popular research topics. It also has influenced a lot of research papers on other languages, including SQL and SBQL. Among the papers on XQL was a proposition of an extension to the XQuery – an inflationary fixed point operator, which is a controlled form of recursion [Afan08, Afan09]. Example 1.3 presents a sample usage of this operator.

Example 1.3 Query that recursively computes all prerequisite courses, direct or indirect, of the course coded with "J1", on an instance document "curriculum.xml"

```
with $x seeded by doc ("curriculum.xml")//course[@code="J1"]
recurse $x/id (./prerequisites/pre_code)
```

2 Stack Based Approach and the Stack Based Query Language

Stack Based Approach (SBA) has been introduced by K. Subieta in [Subi94]. It is a general approach to construction of query languages for object-oriented and semi-structured databases. The main idea of the SBA is to construct query languages in the methodology of programming languages. The languages should combine database support beyond simple querying (updates, views, schema manipulation) with programming abstractions such as variables, functions or classes. Most impedance-mismatch problems would be eliminated by this design. Such language, with proper syntax (and syntactic sugar) could also be an efficient tool for database application development.

In the Stack Based Approach semantics of a query is based on the mechanisms used in programming languages – like the call stack. This approach is compatible with the *naming-scoping-binding* paradigm – each name occurring in a query is bound with a proper entity according to the scoping rules. The actual stack used by the SBA is an extension of a classical call stack. For example it can handle various data collections appearing in structured and semi-structured databases.

SBA relies on the three basic elements: data model, environment stack and so-called non-algebraic operators.

2.1 Data Models

Subieta proposed a set of store models that could be used in the Object DBMS. The basic model is called AS0. It was first defined in [Subi94] and called there the Abstract Data Model M0. The [OMG07] renamed this model to its current name AS0 (Abstract Store Model M0).

In the AS0 data model object states are represented as triples $o=<i, n, v>$ where i is the object identifier, n is its name, v – its value. Each identifier i is unique. Objects are divided into three categories; the division is based on the type of v:

- If v is an atomic value (e.g. number, string, Boolean value) then the object o is called an

atomic object.

- If *v* is an objects identifier *i* (of an object stored in database), than the object *o* is a **pointer object.**
- If *v* is a set of object states than the object *o* is a complex object.

Objects of the same name may be of different types or may contain different amount of sub-objects. In the model AS0 an object store is a pair (O,R) where O is a set of object states and R is a set of the identifiers of the top-most objects (roots). Each object in the store should be reachable from root objects by either pointers or parent-child relations.

Example 2.1 A simple database:

```
<i1, Emp,{ <i2, fname, "John">, <i3, sname, "Smith">,
    <i4, dept, i17>, <i5, salary, 2000> }>
<i6, Emp,{ <i7, fname, "Bob">, <i8, sname, "Gordon">,
    <i9, dept, i17>, <i10, salary, 2300> }>
<i11, Emp,{ <i13, sname, "Watson">, <i14, dept, i22> }>
<i17, Dept,{ <i18, name, "IT">, <i19, employee, i1>,
    <i21, employee, i2>, <i37, boss, i6> }>
<i22, Dept,{ <i31, name, "administration">,<i33, boss, i11> }>
R=[i1,i6,i11,i17,i22]
```

Besides AS0 model, Subieta introduced more advanced models called AS1, AS2 and AS3 that extend AS0 with object oriented features. The simplest of them is AS1 model. It is basically the AS0 model with support for classes and inheritance. It is obtained by adding of a set of classes' identifiers C and two relations: CC that determines inheritance among classes, and OC that determines the membership of objects in classes. Classes are stored as complex objects, yet their identifiers do not belong neither to O, nor R. The CC relation cannot contain cycles, however AS1 model allows multiple inheritance. AS2 and AS3 models add features such as interfaces, roles and encapsulation.

2.2 ENVS

The concept of the *call stack* (*environment stack*) appeared in the 1950's when the first compilers of the high-level languages where created. Since then it has become the basis of many programming languages like Pascal, C, Java, Python, etc.

In programming languages the concept of *environment* of program execution denotes all the run-time entities (variables, constants, objects, functions, procedures, types, classes, etc.) that are available at the given point of the program control. The environment is a structure that changes during the execution of a program. It consists of sub-environments that appear and disappear during the run-time.

The call stack is a main memory structure that is assigned to a single client application program or to a single process or thread. A section is associated with a particular *procedure call* or an executed *program block*. When the control is shifted to a procedure call, a new section with all entities local to this call is pushed onto the top of the stack. The section is popped from the stack when the procedure or program block is terminated. For the procedure that is currently running all values of parameters, local variables/objects and any other local entities are stored within the top stack section.

In SBA the environment stack (ENVS) is an extension of the standard call stack. It has some additional functionality that concerns name binding (which implies the search on the entire stack), scoping rules (skipping some sections) and in rare cases inserting new sections in the middle of the stack. ENVS consists of sections (environments). Each section contains a collection of zero or more entities called the *binders*. A *binder* is a pair (*n,x*) (usually written *n(x)*), where *n* is an external name of an entity, and *x* is a value of this entity. Binder's value may be an atomic value, an identifier or even a collection of values.

The two most important operations involving ENVS are *name binding* and *object nesting*.

2.2.1 Name Binding

Binding of a name is a compiler/run-time action of acquiring a value of an entity using its name. This task is performed using ENVS. The general method is to search for a binder with a

given name inside of the environments, starting from top of the stack and proceeding to the bottom according to the scoping rules. The search stops when an environment containing at least one matching binder is found. More formally the name binding is performed according to the following steps:

- Check the top section of the stack for a binder named *n*;
- If the checked section contains a non-zero number of binders named *n*, the result of binding is the bag of all values stored within these binders
- If the top section does not contain such binder, next section is checked.
- Such process is continued in lower and lower stack sections, until a binder named *n* is found or there are no more sections left.

The above algorithm is sufficient for the AS0 model. In more complex data models visiting particular stack sections is governed by advanced *scoping rules* that require omitting some sections.

2.2.2 Nesting

Nesting is an operation of creating a new section of ENVS containing binders to the interior of an object or a procedure. In the SBA it is done with the help of the *nested* function. This function takes any query result as an argument and is implicitly parameterized by an object store. For the argument it creates a set of binders. This set should then be pushed as a section at the top of ENVS. The function *nested* is exhaustively explained in [Subi04]. In general, depending on the argument, the result of the function call *nested(i)* is:

- If i is an identifier of a complex object $<i, n, \{ <i_1, n_1, \ldots >, \ldots , <i_k, n_k, \ldots > \}>$, $nested(i) = \{ n_1(i_1), n_2(i_2), \ldots , n_k(i_k) \}$.
- If i is an identifier of a pointer object $<i, n, i_1>$, and the object store contains the object $<i_1, n_1, \ldots >$, then $nested(i) = \{ n_1(i_1) \}$.
- If i is a binder $n(x)$ then $nested(i) = \{ n(x) \}$.
- If i is a structure **struct**$\{ x_1, x_2, x_3, \ldots \}$, then *nested(i)* returns the union of the results of

the *nested* function applied to the elements of the structure: *nested*(i) = *nested*(x_1) ∪ *nested*(x_2) ∪ *nested*(x_3) ∪ ...

- For other arguments the result of *nested* is the empty set.

The function *nested* only returns a set of binders to be placed on the ENVS, but it does not open a new section on the ENVS itself. It should be done by the query execution engine.

Another function connected with the object nesting – it is the *pop()* function. Its functionality is similar to standard pop functions associated with stacks, however SBA's *pop* function has broader functionality. For example, it performs a check for binders of temporary objects and deletes such objects from temporary store unless they are referenced elsewhere. Example 2.2 presents the result of nesting of an Emp object form the example 2.1

Example 2.2 Nesting of a complex Emp object.

```
<i6, Emp,{ <i7, fname, "Bob">, <i8, sname, "Gordon">,
     <i9, dept, i17>, <i10, salary, 2300> }>
```

nested(*i6*) = { *fname(i7), sname(i8), dept(i9), salary(i10)* }

2.3 SBQL

The Stack Based Query Language (SBQL) is a prototype object query language realizing the Stack Based Approach [Subi94, Subi04]. It is the model language for all other projects influenced by SBA. The basic idea of SBQL is to combine querying and programming capabilities in one language that eliminates impedance mismatch. SBQL's query semantics is based upon recursive evaluation of the syntax tree and binding of names using environment stack.

The first version of this language was implemented in the LOQIS system [Subi90a, Subi90b]. Since then SBQL has been implemented in a number of systems, including European projects like eGov Bus or VIDE. It was also the model language for a number of research papers on design and optimization of query languages. SBQL was also considered as a foundation for the new standard for object-oriented query languages by the OMG group [OMG07]

SBQL has been designed so that it may work with a number of data models, but its semantics is best explained using AS0 model. The AS0 model originally did not include types beside the general division into atomic, pointer and complex objects. However, the works [Sten06, Lent06] propose a semi-strong type-checking method for object query languages. Those works especially apply to SBQL.

2.3.1 Query results and *eval* function

In SBA it is assumed that queries never return objects but references to objects, sometimes within more complex structures. Objects live in the object store; no entity called an object occurs elsewhere. Queries can also return values stored in objects and values calculated by some functions or algorithms. Similarly to other approaches, SBA introduces structures, bags and sequences as results of queries. In formal descriptions of the evaluation process, intermediate and final results are stored on a special stack called the *Query Result Stack* (QRES). The final result placed on the QRES has to be consumed by some agent within the application software, for example by a user interface or by other queries. After a query evaluation is complete, the top of the QRES contains the result of this query. In general a result of a query may only be:

- an atomic value or an identifier (reference) of an object or a stored programming entity like function or a procedure
- a binder, in this context also called a *named value*
- a *structure* of results (*struct*{ x_1, x_2, x_3,... }) (*result* definition is therefore recursive). The order of elements in a structure is significant. A structure may contain values of different types. Also, two named values with the same name are allowed. Structures having no elements are not allowed. Structures are actually similar to tuples, known from relational systems.
- a *bag* and a *sequence* of results are also valid results themselves.

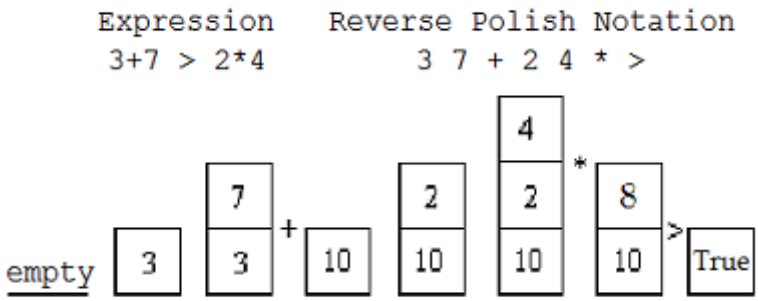


Figure 2.1: States of the QRES during evaluation of the given arithmetic expression

Figure 2.1 present the states of the QRES during the evaluation of the query *3+7>2*4*. Results of the evaluation of numbers 3 and 7 are directly placed on the QRES. Evaluation of the + operator consumes both values and places a new value on the QRES (literal value 10). Further evaluation results in placement of the literal value of Truth on the top of the QRES. This value is the result of the initial query and may be consumed for example by an outer agent.

The semantics of an SBQL query are best explained using its evaluation process. The *eval* procedure described in [Subi04] operates on three structures: the object store, the ENVS stack and the QRES stack. This procedure takes as an argument a query in a form of an Abstract Syntax Tree (AST) and during recursive calculation generates query's final result, which then is placed at the top of QRES. During calculation it may modify ENVS but if a query has no side-effects, the final state of ENVS is equal to the initial state.

The *eval* function is compositional – the result of a query is a direct function of the results of the immediate subqueries. The simplest types of queries are names and literals. The result of the evaluation of a literal is a value of this literal placed in the new section on the top of the QRES. To evaluate a name, it needs to be bound with values (according to the procedure described in the chapter 2.2.1) and the multiset of those values should be placed in the new section on top of the QRES.

The operators in SBQL are divided into two groups: algebraic and non-algebraic operators.

2.3.2 Algebraic Operators

Algebraic operators do not require ENVS to calculate their result out of the partial results of

subqueries. If the operator Θ is an algebraic operator, than to evaluate a query `q1 Θ q2` we need to evaluate independently both subqueries (q1 and q2) and then to process the partial results according to the specific functionality of the Θ operator.

The algebraic operators are well known from programming languages – they include arithmetic operators, comparison operators, aggregate functions or conditional operators.

Example 2.3. Evaluation schema for binary non-algebraic operator Θ

```
def eval(q):
    ...
    case q is qleft Θ qright:
        eval(qleft);
        tl = QRES.pop();
        eval(qright);
        tr = QRES.pop()
        QRES.push( apply_Θ(tl,tr) )
    ...
```

Example 2.3 presents a schema of evaluation for the non-algebraic operator Θ. Push and pop functions are the standard stack operations. Function $apply_\Theta$ depends on the actual operator being executed. It represents the process of calculating the final result using intermediate results calculated earlier. This function does not involve operations on the environment stack and does not depend on the state of the ENVS.

2.3.3 Non-algebraic Operators

The core of SBQL are so called non-algebraic operators. They are binary operators that modify the ENVS during evaluation. Their eval procedure is much more complex than the one for algebraic operators. During evaluation the first step is to evaluate the left subquery. Then for each element *e* from the acquired result collection perform nesting of this element and evaluation of the

right subquery. Next incorporate the acquired result into temporary result set according to the operator's specifics. Finally remove the top environment from ENVS. When every element has been processed push the temporary result set onto QRES.

Example 2.4. Evaluation schema for binary non-algebraic operator Φ

```
def eval(q):
    ...
    case q is qleft Φ qright:
        eval(qleft);
        tl = QRES.top();
        for each r in tl:
            ENVS.push(nested(r))
            eval(qright);
            tr = QRES.pop()
            partialresult[r]:=combine_Φ(r, tr)
            ENVS.pop()
        QRES.pop()
        QRES.push( merge_Φ(partialresult))
    ...
```

Example 2.4 presents a schema of evaluation for the non-algebraic operator Φ. Top function returns the topmost value of the stack without removing it. Function $combine_{\Phi}$ and $merge_{\Phi}$ depend on the actual operator being executed and they do not depend on the state of the ENVS. Detailed information on those functions in context of each of the non-algebraic functions may be found in [Subi04].

Among the key non-algebraic operators for SBQL are the selection operator (*where),* projection and navigation operator (*dot*)*,* navigational join and quantifiers. Examples 2.5 and 2.6 show a sample usage of dot and where operators, while example 2.7 presents a schema of evaluation of a query that includes the non-algebraic operator *where*. During this evaluation we assume that the database has the structure from the example 2.1

Example 2.5. For each employee working in the ‘IT’ department return their personal data and salary

(Emp where deptname=='IT').(name+' '+surname, salary)

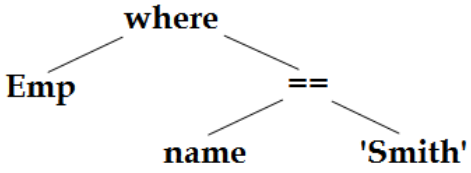
Example 2.6. For each department return its name and the average salary paid to its employees

Dept.(name, avg(worksIn.Emp.salary))

Example 2.7 The schema of evaluation of a query based on the data from the Example 2.1

I. Query: Emp where sname == 'Smith'

AST:



Initial state of ENVS:

$Emp(i_1)$ $Emp(i_6)$ $Emp(i_{11})$ $Dept(i_{17})$ $Dept(i_{22})$

Initial state of QRES: *empty*

II. The root operator is the non-algebraic operator *where* thus the first step is to evaluate the left subquery: the name *Emp*

ENVS	QRES
$Emp(i_1)$ $Emp(i_6)$ $Emp(i_{11})$ $Dept(i_{17})$ $Dept(i_{22})$	i_1 i_6 i_{11}

III. Now, according to the evaluation schema, for each of the identifiers placed on QRES we need to evaluate the right subquery: s*name* == *'Smith'* , where the comparison operator belongs to the group of the algebraic operator. First we should *nest* the complex object identified by i_1, and then to evaluate the name s*name* and the literal *'Smith'*. We will do this in one step:

ENVS

fname(i_2) sname(i_3) dept(i_4) salary(i_5)
Emp(i_1) Emp(i_6) Emp(i_{11}) Dept(i_{17}) Dept(i_{22})

QRES

'Smith'
i_3
i_1 i_6 i_{11}

IV. To evaluate the comparison operator we pop two sections of the QRES and according to the specification of the equality operator, we check if the atomic object identified by i_3 (<i_3,sname,'Smith'>) holds the atomic value 'Smith'. We push the result of comparison on the QRES:

ENVS

fname(i_2) sname(i_3) dept(i_4) salary(i_5)
Emp(i_1) Emp(i_6) Emp(i_{11}) Dept(i_{17}) Dept(i_{22})

QRES

True
i_1 i_6 i_{11}

V. The *where* operator consumes the *True* value from the QRES and adds the identifier i_1 to the partialResults collection. The top section of the ENVS is removed and the next object (i_6) is nested on the ENVS. The steps III and IV are repeated:

ENVS

fname(i_7) sname(i_8) dept(i_9) salary(i_{10})
Emp(i_1) Emp(i_6) Emp(i_{11}) Dept(i_{17}) Dept(i_{22})

QRES

False
i_1 i_6 i_{11}

partialResults

{ i_1}

VI. Because this time the comparison operator returned the *False* value, the i_6 object is not included in the partialResults collection, although the top section of the QRES is removed. Again the top section of the ENVS is removed and the next object (i_{11}) is processed according to the steps III – V. For this object the comparison operator would evaluate into the *False* value, thus this object is not included in the partialResults. The top section of ENVS is removed. Because all the identifiers from the QRES have been processed, again the *pop*() function is called on the QRES leaving it empty:

ENVS	QRES	partialResults
Emp(i_1) Emp(i_6) Emp(i_{11}) Dept(i_{17}) Dept(i_{22})	*empty*	{ i_1:i_1}

VII. Now we merge the partialResults according to the requirements of the *where* operator. The result of this operation is then pushed on the QRES. The result of the evaluation of the query is:

ENVS	QRES
Emp(i_1) Emp(i_6) Emp(i_{11}) Dept(i_{17}) Dept(i_{22})	**i_1**

In SBQL recursive queries are available through the use of the non-algebraic operator *close by* and its variants: *leaves by, close unique by, leaves unique by*. This operator's syntax is simple:

query$_1$ `close by` *query*$_2$

Both left and right query have no other restrictions to their structure besides that they should return a bag of results of the same type. The left query is the seed query – it provides the initial bag of elements. The right query is executed recursively in the context of each element from the acquired result bag for each recursive step. The calculation stops when the execution of the right query returns an empty bag. An example of a close by query is:

(Employee where name = "John Smith") close by (boss.Employee)

This query finds all employees that are John Smith's direct or indirect superiors. But sometimes we would like to get only the leaves of a result tree – in the example above only those employees that do not have superiors. In such cases we may use another recursive operator called *leaves by*. Unfortunately both operators *close by* and *leaves by* suffer from the same disadvantage – they may create infinite loops. In such cases other recursive operators may be applied – *close unique by* and *leaves unique by*. Those operators are variants of the *close by* and *leaves by* operators that remove duplicates on the fly after each closure iteration. This way they eliminate loops. The above semantics is similar to the Delta semantics of the XQuery's *seeded by* operator [Afan09] mentioned earlier.

Close by operator represents transitive closure. The SBQL language is also equipped in a

mechanism of fix-point operations that may be applied to a vast range of queries. The operator that fulfills this mechanism is *fixpoint.* The general mechanics of *close by* and *fixpoint* are similar. Their functionality is exhaustively presented in [Subi04, SBQL]. The SBQL language also provides the mechanism of recursive views and recursive procedures. Their description may be found in [Piec06]

2.3.4 SBQL Implementations and Language Comparison

SBA (Stack Based Approach) and SBQL has been implemented in number of research and business projects. The first business project to implement a language based on the SBA was Netul developed in 1989 by Intra-Video. A year later an experimental project Loqis has been started [Subi90a, Subi90b]. This project was the first to implement the language SBQL. Among other implementations are VPOS – a query language for the XML DOM model, LoXiM and Monad/PySBQL database systems. SBQL was also a key element of the following European projects: ICONS (Intelligent Content Management System), VIDE and eGov Bus.

In February 2006, the OMG announced the formation of the Object Database Technology Working Group (ODBT WG) to develop the "4th generation" standard for object databases. This group is planned to continue (in a way) the work of ODMG. ODBT WG considered SBQL language [OMG07] and LINQ framework as the background for this standard. The work of this group is still in progress.

The following examples compare the syntax of SQL, OQL, XQuery and SBQL languages. The examples assume that a database stores a simple schema containing information on students. Each student has a name, information about the year of study he attends.

Example 2.8

Description	Get full information on students
SBQL	`Student`
OQL	`Student`
SQL	`Select * from Student`
XQuery	`doc(...)//Student`

Example 2.9

Description	Get the names and year numbers of all students
SBQL	`Student.(name, year)`
OQL	`Select name, year from Student`
SQL	`Select name, year from Student`
XQuery	`for $s in doc(...)//Student` `return {$s/name, $s/year}`

Example 2.10

Description	Get the names of all 1-st year students
SBQL	`(Student where year = 1).name`
OQL	`Select name from Student where year = 1`
SQL	`Select name from Student where year = 1`
XQuery	`doc(...)//Student[year=1]/name`

2.4 PySBQL as a Testing Platform

PySBQL language [Burz07] is an attempt to reconcile the concept of Stack Based Approach in databases [Subi94] with construction of Python [Python] language which is a popular programming language. Stack Based Approach in databases, as I have tried to show in previous chapters, gives a good start for development of a computer language that, at the same time, is a programming and an object query language. The joining of SBA with Python resulted in a language, in which writing database applications became simple and clear.

My design of the initial syntax and semantics of PySBQL was published in [Burz07]. The first prototype of this language was implemented in Java and worked with AS0/XML storage. The goal of this implementation was to develop a multi-platform language free of impedance mismatch problems. Since then this project has undergone many changes. The second prototype was intended for distributed object database and was code-named "Monad/PySBQL". I have implemented the latest prototype in Python language in 2009. This version will be described in the following chapter. It is designed to work with AS0 and XML data models, and was a testing platform for rewriting algorithms, query processing on Graphical Processing Units (research in progress) and data storage

models (research in progress).

The grammar has been coded using PLY [PLY] library for Python. This chapter presents core fragments of PySBQL in a manner similar to BNF notation. In PySBQL literal values and variable names are defined after Python. The other grammar constructs adapted from Python are indent based syntax, loop statements (for, while), *if-else* statement, function definition's header. The table 2.1 presents the main grammar rules.

query ::= literal | name | (<newline>)* query (<newline>)*	
blockQ ::= <newline> <indent> (query)* <dedent>	Block of queries
query ::= **unaryOperator** query	Unary algebraic operators
unaryOperator ::= + | - | ~ | **not**	
query ::= query **binaryAlgOperator** query	Binary algebraic operators
binaryAlgOperator ::= compOp | boolOp | arithOp | bitOp	
compOp::= == | < | <= | >= | > | **is** | **is not** | **in** | **not in**	Comparison operators
boolOp::= **and** | **or** | **xor**	Boolean operators
arithOp::= + | - | * | / | // | % | **	Arithmetic operators
btOp::= & | **|** | ^ | << | >>	Bitwise operators
query ::= query **where** query | query **.** Query | query **join** query	
query ::= query assignOp query | query **<-** query	
assignOp ::= <- | = | += | -= | *= | /= | //= | **= | %= | &= | |= | ^= | >>= | <<=	
query ::= **rename** query **as** name	Name definition
query ::= **new (temporary | local | permanent)** name : ("**(**"name **:** query (**,** name **:** query)* "**)**" | literal)	Creating a new object
query ::= **ref** query **as** name	Creating a local pointer object
query ::= **deref** query	Dereference on query
query ::= query **group as** name	Grouping and name definition

Table 2.1: Main syntax rules for PySBQL

2.4.1 Examples of PySBQL Queries and Programs

Example 2.11 Obtain the names and birth years of chosen employees' children. The "\" character is interpreted as a line break that allows for multiple line statements

```
(employee where job_record.job_date>'2000-01-01').children. \
     (child_name, birth_year)
```

Example 2.12 Give each employee a raise by 100:

```
employee.salary+=100.0
```

Example 2.13 Rename the field *town* in the *address* objects into *city*

```
rename address.town as city
```

Example 2.14

Create a new complex object *company* containing one atomic object named *name*, one pointer object *located_in* and one complex object *manager*

```
new company : (name : 'TransCom',
     located_in : city where name == 'London',
     manager : (name : 'Alan Willson',
               phone : "644-77-99") )
```

Example 2.15 Two versions of program that prints on standard output the names of all employees

```
print employee.name
```

or:

```
for e in employee:
     print e.name
```

Example 2.17 For each person write their name and the social title Mr. or Mrs. depending on person's gender

```
for p in person:
     if p.gender == 'f':
          p.name = 'Mrs. ' + p.name
     else:
          p.name = 'Mr. ' + p.name
```

Example 2.18 Definition and a sample usage of a simple factorial function that would be stored in a database as an complex object

```
def permanent factorial(a = 0):
     i,k = 1, 1
     while (i<a):
          i+=1;
          k*=i
     return k
print factorial(4)
print factorial()
```

2.4.2 PySBQL vs Python

Just like most of the query languages, Python is a dynamic, interpreted and interactive language. Other key features that influenced the decision of developing PySBQL based on Python's syntax were:

- clear, readable syntax resulting in easiness in learning and using
- high level dynamic data types
- embeddable within applications as a scripting interface

- Python is equipped with some aspects of a functional language (like lambda expressions)
- popularity – a language that would be very close to Python would be easy to master for Python's programmers.
- Python's standard implementation is under an open source license that makes it freely usable and distributable, even for commercial use. Based on the modules of Python, new modules for PySBQL could be developed in a short time
- it is an interesting subject to study how far a query language based on SBA can be integrated within an interpreted language

Although PySBQL and Python share much of their syntax, their functionality differs. The basic difference between Python and PySBQL is management of persistent data, construction of a call stack, modified assignment statement to cope with persistence, and mostly – the evaluation process.

On the other hand, the idea behind PySBQL was to firstly establish a solid base for data management, thus the research on this language concentrated around database access optimization and optimization of query processing.

2.4.3 PySBQL vs SBQL

Database management in PySBQL was based on the research on SBQL language. Both languages share the construction of non-algebraic operators, however the SBQL's prototypes have been greatly influenced by syntax of languages like Java and C#. Initially PySBQL project was a study on effects of combining an interpreted dynamic language with Stack Based Approach.

The differences between PySBQL and SBQL involve handling of data collections (sequence, bag and structure in SBQL, list and dictionary in PySBQL), interpretation of non-Boolean values as true or false when logic value is needed, handling of variables and much of additional syntax.

SBQL implements semi-strong static typing [Sten06], while PySBQL has dynamic type system with type checking based solely on variable values and not with variable names. This means that we may dynamically assign different values with different types to the same variable, however type

checking would return a type error when trying to subtract list-typed variable from integer-typed. This is directly connected with the basic approach to variables themselves: contrary to SBQL, variables in PySBQL are simply binders and not atomic objects.

Some of additional differences appear evaluation process on the base implementation level, however since both languages are in their prototyping stages, those differences will be omitted here.

2.4.4 Left and Right Dereference

One of the differences between PySBQL and SBQL is the approach to the `deref` operator. In SBQL dereference operator is inserted automatically during the generation of a query's AST. However there is no clear convention of autodereference.

One of the examples, when names are bound in different contexts depending on which type of value is needed is the function call `f(a=b, b=a)`. In this call the names `a,b` from the right sides of the assignment operators are bound according to the context of the function call, whereas `a,b` from the left sides of those operators are bound in the context of the interior of the function, the parameter list to be exact.

This problem and the unclear approach to dereference in the SBA became an inspiration for me to develop a new approach based on a classical concept of l-value and r-value for the name of a variable\object. This approach is more universal – it is designed to deal with cases when in some context the same name may represent different variables\objects depending on whether we ask about l-value or r-value. Results of this study were published in [Burz09a]

2.4.5 L-values and R-values in PySBQL Language

The new approach to variable dereference is based on two mechanisms: l-binding and r-binding (left and right binding) of names. Each of those mechanisms is equipped with its own environment searching rules. L-binding and r-binding for algebraic operators and imperative constructs are consistent with the convention of Python language. Only object and the non-algebraic operators need a detailed discussion.

For an object its l-value means the reference to this object. Particularly if a binder built from name and value indicates some object, then the value of this binder is always the address of this object. In this sense our concept corresponds to the concept of the ENVS for SBQL language. R-values, being the values of objects, are more distinguished.

For a pointer object its r-value is an address contained within such object. The r-value of an atomic object is the content of its value field. For a complex object, the r-value is a collection of binders to its subobjects with added binder named `self`, which value is a reference to this object.

Non-algebraic operators expect on their left side a collection of r-values of complex objects. Their further evaluation depends on whether it is evaluated with respect to l-value or r-value.

When the r-value is expected from the `where` operator, it returns a collection of r-values of the result collection. When the l-value is expected – the `where` operator returns proper l-values of the left subquery, filtered by the right subquery. The `dot` operator evaluates the right subquery passing the information whether the l-value or r-value is needed. The evaluation takes place for each r-value returned by the left subquery. The collection of acquired results passed as a result of the `dot` operator's call.

Among other non-algebraic operators are quantifiers and *close by* operator. Both quantifiers require r-values of their first subqueries, and then for each r-value they evaluate the second subquery. Existential quantifier returns a positive result if the second query returned a positive result at least once. In other cases it returns a negative result. The universal quantifier returns false if the second query was at least once evaluated to a negative result. The *close by* operator evaluates the r-value of a first query. For each value gathered in this stage it is nested on the ENVS and the evaluator requests an r-value of the second query. Collection of r-values gathered this way is added to the partial result collection and used to repeat this evaluation step. The process is repeated until the second query returns an empty collection.

Of course there are special cases for which there is an explicit need to enforce value or address. Here we allow for explicit usage of *ref* operator for fetching reference and *deref* (or @) operator for fetching value. Their usage cannot be redundant. Usage of *deref* operator where a literal value appears will result in a Dereference Error. This operator's purpose is for example to fetch a value of an object pointed out by a pointer object, however it may be also used with atomic and complex objects

3 General Strategies of Optimization by Rewriting

There are many methods of query optimization. Among them is query rewriting. It is usually one of the initial phases of query processing. This method is based on the notion of query semantic equivalence. Two queries are semantically equivalent if they produce the same results regardless of the database state. The equivalence of queries is the subject to a set of rules. The most important may be found in [Ullm88, Denn91]. Query rewriting comprises a number of transformations of the original query whose goal is to produce an equivalent query that has shorter evaluation time or consume less system resources. Such transformations do not depend on the physical state of the system. However they may require access to schema information. The most common rewriting transformations are:

- subquery un-nesting and flattening
- views and functions inlining
- early selection/projection by predicate move around
- query merging
- rewriting to other language/algebra/monoid comprehension calculus

One of the strategies of query evaluation and optimization is to parse and rewrite a query into a corresponding syntax tree according to the grammar rules of the given language. Such tree of a query maybe then used as an input of query optimization algorithms.

The amount of available query optimization algorithms is huge. To describe them all in detail one could write a multi-volume encyclopedia. The following chapter presents a selection of research on rewriting algorithms from those groups. Selected techniques have either inspired or closely relate to the research on the algorithms presented in the chapters 4 and 5 of this thesis.

3.1 Optimization of Non-recursive Queries

Most of the queries to modern database systems are non-recursive. Their variety results in a large amount of optimization methods that serve different purposes and are applicable at different

stages of query processing. The following sub-chapters present seven general methods of optimization by rewriting that may be applied to non-recursive queries.

3.1.1 Predicate Move-around

Predicate move-around is a commonly used optimization technique. It has been described by Levy et al. in [Levy94]. This technique is a generalization of a similar, well-known technique – predicate push-down [Ullm88]. Predicate push-down allows for early selection by pushing selection predicates down the tree of a query. Predicate move-around optimizes queries by firstly moving predicates up the query tree before pushing them down into the subqueries or views they refer. This way predicates pulled up from one query block can be pushed down into another block. The original paper, influenced by previous research on moving predicates, discusses situations where rewriting two query blocks into one ([Hell92]) is either impossible or complicated, yet predicates can be moved. Those situations include aggregate views/subqueries. Other advantage of predicate move-around is that it may be applied to a variety of predicates including string comparisons and existence predicates.

Another similar algorithm is described in [Yan94]. It is based on performing the group-by operation before joins in order to reduce the size of data processed during joining operation.

An adaptation of predicate pushing methodology for SBQL queries has been described in [Plod00]. It is based on distributivity property of non-algebraic operators such as selection, navigation or join operators. Pushing selection for SBQL may also be viewed as a simplified version of factoring out independent subqueries.

For XPath/XQuery a number of optimization techniques based on predicate move-around have been developed. The paper [Grin05] by Grinev and Pleshachkov describes a technique called predicate push down XML element constructors. Its basic idea is to change the order of operations to apply predicates before XML element constructors. A set of rewriting techniques for XQuery based on predicate move-around has been discussed in [Ozca08]. They are called XPath pushdown, local predicate pushdown and join pull up. All those techniques work with so called local predicates – predicates and simple XPath navigation queries that access only one document. XPath pushdown considers navigational steps as existential predicates. It involves rules to push down XPath through operations such as selection, set union and XML element construction. Pushing down conditional selection predicate into an XPath expression is the main aspect of the local predicate pushdown

technique. Application of this technique is presented by the Example 3.1. The next technique – join pull up, also called simple decorrelation, works with join predicates embedded in XPath expressions and pulls them into the where clause.

Example 3.1

Original program:

```
for $c in db2-fn:xmlcolumn("sample.doc")/c, $a in $c/a
     where $c/d = 5
     return $c
```

Transformed program where predicate `$c/d = 5` was push down:

```
for $c in db2-fn:xmlcolumn("sample.doc")/c[d = 5], $a in $c/a
     return $c
```

3.1.2 View/function Inlining and Merging Nested Subqueries

Presence of user-defined functions or view calls in a query may cause the optimizer to work less efficiently. This problem may be at least partly solved using function inlining. It is a common optimization technique used for example in programming languages' compilers. The basic work on view inlining is [Ston75]. When dealing with non-recursive views or functions this technique is simple. However, when dealing with recursive user-defined functions there exists a possibility of generating an infinite loop. This problem in context of XQuery language has been addressed by Grinev and Lizorkin in [Grin04]

View/function definition expansion and inlining may be a first step of merging nested subqueries. There is a lot of research on un-nesting of correlated nested SQL queries and merging them into a single query. Among the most important are [Kim82, Daya87, Gans87, Mura92]. The work [Chau98] presents a very good overview on the subject. The following example of subquery merging technique comes from this paper.

Example 3.2

In the following queries EmpNo and DeptNo columns are the primary keys of tables Emp and Dept respectively

Original query:

```
SELECT Emp.Name FROM Emp
 WHERE Emp.DeptNo IN ( SELECT Dept.DeptNo FROM Dept
      WHERE Dept.Loc='Denver' AND Emp.EmpNo = Dept.Mgr )
```

Transformed "flattened" query

```
SELECT E.Name FROM Emp E, Dept D
 WHERE E.DeptNo = D.DeptNo
      AND D.Loc = 'Denver' AND E.EmpNo = D.Mgr
```

This paper discuses more complicated cases of nested subqueries including occurrences of aggregates, quantifiers. The research on merging of XQuery queries has been greatly influenced by the research on merging of SQL queries. The work [Ozca08] discusses two techniques of merging XPath expressions.

3.1.3 Finding Independent Subqueries and Query Un-nesting

Nested subqueries may significantly reduce evaluation efficiency since they usually involve nested-loop evaluation. When a subquery occurs in a main query more than once, it might be profitable to calculate such expression in advance. Also when a subquery placed within a loop does not depend on the controlling variables, it would be preferable to evaluate this subquery only once (for example on the first entry into the loop). Such approach is called query un-nesting or subquery decorrelation [Gans87]. Query un-nesting itself may not result in performance improvement [Fega98]. Instead it allows for further optimization. In literature there are a number of papers on query un-nesting. The paper [Sesh96] explains the problems of decorrelation and surveys previous papers on the subject. The authors of this paper also propose a technique for decorrelation of SQL

queries by extracting distinct outer references and materializing all the possible values of the subqueries. Such approach, being an extension of the magic-sets technique, is known as the "magic decorrelation rewrite".

The problem of un-nesting SQL queries in presence of disjunction is discussed in [Bran07]. The authors of this paper propose an optimization technique based on the bypass operator first introduced in [Kemp94]. This technique's advantages and disadvantages are also addressed by the authors of [Elhe07].

For SBQL queries the problem of finding independent queries and query un-nesting is one of the basic tasks for optimizers. It has been discussed in a number of papers including [Piec10, Subi04, Plod00]. It is based on checking in which section binding of a given name is performed. If all of the names of some subquery are bound in other sections than the one opened by the root operator of this subquery, then this subquery could be unnested. Thus the analysis of the section numbering is the base operation for this type of optimization. For more detailed information on how this is performed, please refer to [Subi04].

Example 3.3 presents a result of applying basic version of an algorithm for factoring-out an independent subquery. In the original query from this example the subquery calculating Smith's salary would be executed as many times as there are Employee objects. In the optimized query it is executed only once. The presented AST of the original query has been labeled according to the following rule: the non-algebraic operators are labeled with the number of a section they would open, while names are labeled with a pair of numbers: the Environment Stack size, and the number of a section that holds a binder for a given name

Example 3.3

Original query:

```
Employee where Salary =
     ((Employee where Surname="Smith").Salary)
```

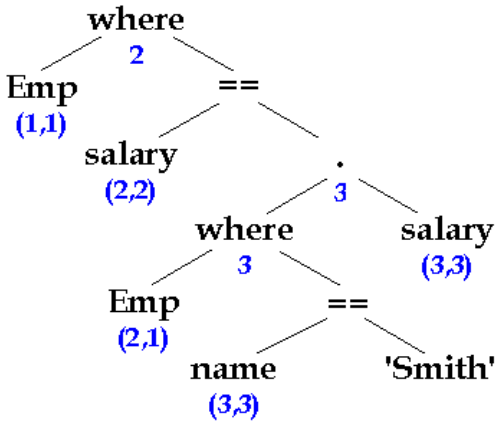


Fig. 3.1: AST of the original query
labeled with section numbering

We observe on the above figure that none of the nodes besides the node labeled *salary* will be bound in the second section. Note that the name *Emp* in the inner selection (dot) clause is bound in the section 1 while the outer *where* operator opens the section 2. Thus the inner selection clause is independent of the outer *where* and can be factored out.

Modified query:

```
(((Employee where Surname="Smith").Salary) group as S).
     (Employee where Salary = S)
```

With the rise of interest in XQuery and semi-structured languages, the issue of query decorrelation has once again attracted attention since correlated queries are common in this language. Most research on un-nesting XQuery expressions is based on algebraic rewriting [May06], however the algebras used for un-nesting usually do not retain the order of nodes. The work [Fega00] by Fegaras and Maier presents an alternative approach that uses monoid comprehension calculus. A selection of un-nesting techniques designed for XQuery can be found in [Norm03]

3.1.4 Rewriting to Other Query Languages

Algorithms that translate queries from one language to another form the widest group among the rewriting algorithms. Their variety is huge and covers many target languages and aspects. In the

past the mostly discussed such translation was from Datalog to SQL [Jeze88, Koym90, Godf94, Deck02]. With the progress of database technology the interest in this transformation significantly dropped, however there is still research being done in this area [Haji05]. This is mostly because of the increased interest in deductive databases designed for storing and analyzing ontologies. Reverse transformations – from SQL to Datalog – have also been interesting for researchers, although not to the same extent. The most detailed paper on this subject is [Bres00]. Datalog has been also a target language for rewriting of XQuery [Alme06, Bene08].

The current popularity of XQuery and established background for SQL databases has lead to a number of research papers on rewriting XPath\XQuery to SQL [Care00, Mano01, Fern02, Deut03, Grus04, Kris04].

The newest trend in research on such rewriting of query languages is based on translating SQL queries to XPath\XQuery programs [Halv04, Jigy06, Vidh10]. While the focus of the paper by Halverson et al. is mainly on querying natively stored XML-data, the paper by Jigyasu et al. describes in detail the actual process of translation.

The mentioned translations gained the most interest among the database researchers, however the variety of such translations is vast. An interesting work by Grust and Sholl [Grus98] describes translation of OQL language to a functional notation resembling Haskell language. The main goal of this translation is to reduce intermediate structures due to application of program fusion algorithm described in the following chapter. For SBQL, the only work on rewriting this language to another one is [Wisl07], which describes the construction of an SBQL-to-SQL mapper.

3.2 Short Cut Fusion for Functional Languages

Intermediate structures are common side effect of the evaluation of programming and query languages. They may have both positive and negative effects on the cost of evaluation. On one hand they may be used to speed up the evaluation, on the other – they may result in unnecessary system resource consumption and indirectly and result in reduced evaluation performance.

In 1990 Philip Wadler [Wadl90] presented an algorithm of elimination of such structures in functional languages which he called the deforestation algorithm. This method became also known as "program fusion" because the basic idea behind it is to "fuse" together two functions of which one consumes an intermediate structure generated by the other. Yet, despite great potential, the

original deforestation algorithm was too complicated and was too restrictive on the input data. This is why many papers on deforestation's enhancements have been prepared. Because of the similarities between functional and query languages, there has been research on adaptation of deforestation algorithms to object query languages [Grus98]. One of the enhancements developed for functional languages is called the *short cut fusion* [Gill93, Joha01] or *cheap deforestation*. It has gained much attention [Jone01, Voig08] mostly due to its effectiveness and simplicity.

Although the original cheap deforestation was developed for functional language called Haskell, this chapter presents the functionality of the short cut fusion technique using the Python language notation. This can be done because Python is equipped with necessary functionality such as *reduce* and *lambda* functions. Python language has been chosen not only because it is a base language for PySBQL, but mostly because its popularity and clarity of notation.

Short cut fusion technique is based on a usage of a collection generating function (`build`) and a rule known as *"reduce/build rule*". This technique is based on a *folding* operation implemented in Python as the built-in `reduce` function, which could be defined as follows:

```
def reduce(fCall, Tlist, init=None):
    if not Tlist: return z
    if init==None and len(Tlist)==1: return Tlist[0]
    return fCall( reduce(fCall,Tlist[0:-1],init), Tlist[-1])
```

Another function needed for short cut fusion is the build function which is defined as follows:

```
build = lambda BinF: BinF(lambda x,y: x+y,None)
```

The `build` function takes as an argument a binary function and applies to it a concatenation function and a special *None* object. The functionality of the `reduce` function is presented by examples 3.4 and 3.5

Example 3.4. Example of usage of the *reduce* function

```
reduce(operator.truediv,[18,2,0.5,5],180)
```

results in: ((((180 / 18) / 2) / 0.5) / 5) which calculates into 2.0

Example 3.5.Example of usage of the *reduce* function

```
reduce(operator.truediv,[18,2,0.5,5])
```

results in: (((18 / 2) / 0.5) / 5) which calculates into 3.6

Now we may proceed with defining the *reduce/build* rule

Definition 3.1. The reduce/build rule

```
reduce(f, build(g), n) == g(f,n)
```

In most of the functional languages the majority of operators and functions can be rewritten into their equivalent compositions of `reduce` and `build` functions. The short cut fusion is applicable to the composition of two functions that could be rewritten into this equivalent. The short cut fusion algorithm firstly performs this rewriting. Having such definitions it applies interchangeably the *reduce/build* rewriting rule and the β-reduction – a standard transformation of the lambda-calculus which may be understood as application of arguments. When none of those transformations can be applied the outer `build` function should be rewritten using its definition. As a result we acquire a function which instead of applying subsequent operations to collections of intermediate data, performs all operations sequentially on individual elements of the initial collection. Example 3.6 presents result of application of short cut fusion.

Example 3.6

Original function definition (concatenates a list of strings into a single string using single space as a separator)

```
merge_lst = lambda ls:concat(map(lambda l:l+' ', ls))
```

Where the *concat* and *map* functions are defined using reduce/build composition as:

```
concat = lambda xs:build(lambda c,n:
          reduce(lambda x,y:reduce(c,x,y), xs, n))
map = lambda f,ys:build(lambda c,n:
          reduce(lambda a,b: c(f(a),b), ys, n))
```

Resulting function definition:

```
merge_lst = lambda ls:reduce(lambda a,b:(a+' ')+b, ls, None)
```

Many papers dealing with various languages and problems have shown that the usage of the cheap deforestation has a positive effect on the speed of program processing and the reduction of system resources consumption. Some languages have different functions for *left* and *right folding*, however the work [Gill96] proves that the short cut fusion does not depend on the choice of *folding* type for the reduce function.

3.3 Optimization of Recursive Queries

Research on recursive queries have been a part of studies on the data querying since the 80s. Recursive queries help to solve problems such as bill-of-material, queries involving corporate hierarchy, finding routes between cities. Naïve evaluation of recursive queries usually is inefficient and consumes too many resources. Most optimization techniques for recursive queries involve modified execution plans, enhanced data retrieval or dynamic procedures [Nejd87]. However, there is a group of rewriting algorithms that optimize the initial execution plans.

The research on SQL's recursive queries have been greatly influenced by Datalog's recursive queries (see [Schn08] for more information). There are a lot of research papers discussing recursion in deductive databases. Significantly less work has been focused on relational recursive queries and there has been very little work in the recent years in this field of studies. However, nowadays the problem of recursion becomes once again popular with database vendors and researchers. The research on optimization of recursive queries may be classified as one of the following trends:

- rewriting represented by the magic-sets technique [Ullm86, Mumi94, Ordo05]
- memoing [Diet87] and storage problems [Ordo10]
- cost models and modified execution plans [Ghaz06]

A comprehensive study on optimization of SQL recursive queries may be found in [Ordo10].

Some of the optimization algorithms for SBQL's close by operator may be found in [Subi94]. However, the most comprehensive study on optimization of recursive SBQL queries is [Piec10] which present many interesting techniques such as rewriting by pushing out selection, factoring out independent queries, detection of non-recursive equations and stratification.

3.3.1 Tail-recursion

Tail recursion (known also as tail-end recursion) is a well-know optimization technique for evaluation of recursive functions [Maie88, Rubi10]. It is widely used in functional programming languages, which often use recursion for computation. It is applicable to cases of recursive functions in which the last operation before returning a result is to perform recursion. Such operation is often called a *tail call*. A special case of tail recursion involves situation where the result of the recursive call is not used. The key idea behind this technique is to replace recursion with iteration to decrease the amount of memory (stack space) used and increase efficiency. In most cases when tail calls occur, there is no need to return the result to the intermediate function call – the newly generated result may be returned directly to the initial function's caller. There are a lot of algorithms that are used to rewrite a recursive function's definition, so that it could benefit from tail recursion optimization.

For query languages, tail recursion has been broadly discussed in the context of optimizing Datalog programs. The papers [Ross91, Ross96] by Ross present special cases of magic templates (see chapter 2.2.2) technique enhanced with tail recursion, while the papers [Rama91, Ullm95] describe more generally the application of tail recursion to Datalog programs.

Another query language that benefits from tail recursion techniques is XQuery [Kay06]. However, because of the availability of FLWOR construction equipped with loops, queries that meet the conditions for tail recursion are rarely used. At the same time, the availability of loop construction allows for application of rewriting techniques that result in tail calls. Such techniques are usually adapted forms of rewriting techniques for programming languages Example 3.7 presents application of rewriting technique used in Saxon XQuery processor. The output is a function that can be optimized using tail recursion.

Example 3.7

Original program:

```
declare function local:before_sum($start as xs:integer) as
xs:integer
{    if ($start eq 0) then 0
     else $start + local:before_sum($start - 1) };
```

Modified program:

```
declare function local:tailcall_sum(
     $start as xs:integer, $acc as xs:integer) as xs:integer
     {
     if ($start eq 0) then $acc
     else local:tailcall_sum($start - 1, $start + $acc)
     };
```

3.3.2 Magic Set Techniques

The magic set rewriting technique has been introduced by Ullman in [Ullm86]. The version presented in that paper is the most widely known "magic set" variant. This variant transformed recursive Datalog programs to gain more efficient evaluation. The common result of magic set transformations is a newly generated program or query that contains additional predicates. Such output queries usually have more keywords and clauses compared to the original query, but their evaluation time is shorter [Ullm89].

Since the original paper by Ullman, many extensions and modifications of the basic algorithm has been proposed. The paper [Beer87] by Beeri and others introduces a technique called "Supplementary Magic Sets". It eliminates some of the repeated computation appearing during query evaluation. Other improvements of the original technique are so-called "Magic templates" and “Alexander templates” [Rama88, Seki89].

The research on optimization of recursive SQL queries has been greatly influenced by magic

set techniques developed for Datalog. Among the first papers that showed application of magic sets to SQL language were [Gupt92, Mumi94]. Those papers actually applied magic sets to non-recursive queries compliant with SQL-92 standard. However, recursive SQL queries are easily represented as Datalog programs, thus magic sets have been naturally adapted to SQL. One of the papers dealing with optimization of recursive SQL queries based on optimization techniques for Datalog is [Bris06].

Magic sets have also found applications for XQuery and XPath queries. Two of the papers on the subject are [Alme06, Ozca08]. The first of those papers also discusses effect of proper indexing on programs transformed with magic set technique. The basic idea of this paper is to translate XPath expressions and source data into Datalog programs.

3.4 Open Problems

A lot of work for syntactical rewriting has already been done. However, there are still open problems that can be addressed, mostly in the field of semi-structured query languages. The SQL language has been available on the database market for a long time, and there has been a lot of research conducted on most of its aspects. Yet, there are still open rewriting problems related to this language. An interesting research topic is rewriting a query to benefit from materialized views either for security purposes or to reduce the query execution time.

The research proposals common for all query languages deal with optimization of user-defined functions. This especially applies to the recursive XQuery and SBQL functions. The solutions could be based on function inlining but would require gathering specific schema information. It could also be worth checking if such functions could be optimized at algebra level.

4 Deforestation of Linear Queries

During the execution of the SBQL queries a lot of intermediate structures are being created which may have a negative impact on the execution time. This was the reason behind construction of a new algorithm reducing the size of intermediate structures that works on the level of execution plans. This chapter presents an extended version of the initial version of the algorithm presented in [Burz10].

4.1 Simple SBQL Query Deforestation

The main idea behind this algorithm is inspired by a similar work for OQL [Grus98] and the shortcut fusion algorithm described in the chapter 3.1.5. The execution plans in the following sections are written using lambda expressions from Python language and are represented using Abstract Syntax Trees (AST). The *reduce/build* rule of the shortcut fusion algorithm was also the base rule for the algorithm described in this chapter. Let us remind it.

Rule 4.1. Basic reduce/build rule:

For all two-argument functions `f` and `g` every occurrence of the function call:

```
reduce( f,( build(g) ), n)
```

may be replaced with `g(f, n)`

Application of shortcut fusion to SBQL requires three steps. The first is to create a proper definition of the *`build`* function without violation of the main concept. The second is to create execution plan in the reduce/build notation for each operator. While doing it we must include the operations on the Environment Stack. The last step takes place during the creation of an execution plan for a composite query. It consists of interchangeable application of reduce/build rule with λ-calculus conversions until no more transformation can be used.

Let us start by defining a proper *build* function and preparing a new set of execution plans:

Definition 4.2. The *build* function

```
def build( f ):
    return f(struct.__add__, struct() )
```

The following table presents execution plans for five main SBQL operators:

`where = lambda q1,q2:build( lambda c,n:reduce((lambda ys,y:\` `(nested(y), (q2 and c(ys,y) or ys), pop())[1]),q1,n))`
`dot = lambda q1,q2:build( lambda c,n:reduce((lambda ys,y: \` `(nested(y), reduce(c,q2,ys), pop())[1]), q1,n))`

`join = lambda q1,q2:build( lambda c,n:reduce((lambda ys,y: \`
`(nested(y), reduce(lambda e es: \`
`c(es,struct(y,e)),q2, ys), pop())[1]), q1,n))`

`all = lambda q1,q2:build( lambda c,n:reduce( (lambda ys,y:\` `y and (nested(y), q2, pop())[1]) , q1,True))`
`sum = lambda q1: reduce((lambda ys,y:__add__(ys,y)),q1,0)`

Table 4.1: Execution plans' definitions

The *all* operator can be used for expressing functions like *forall*, *exists*. Also, the *sum* function may be used as a prototype for functions like *count, min*, etc. Having the above definitions we also need rules for rewriting the Abstract Syntax Trees (AST) of execution plans for the input queries.

The definition 4.2 was the basis for the creation of the following rule of AST transformation:

Rule 4.2 *Build* function expanding

Let T be an AST with a root node R labeled "*build*". Let R have a single child that is a subtree S. The tree T may be rewritten into an equivalent AST in which:

- the root node, labeled "function_call", has two child nodes;
- the first child of the root node is the subtree S
- the second child node is labeled "parameters" and has two child nodes: the first labeled "struct.__add__" and the second - "struct()"

The next rule is an adaptation of the basic reduce/build rule (4.1) to the AST of a query execution plan.

Rule 4.3 Reduce/build rule for execution plan's AST.

Let Q be an AST representing an execution plan and S be a subtree such that:

- its root node R is labeled "reduce"
- the second child node N of the root node is labeled "build"
- tree T1 is the first child of R and tree T3 is the third child of R

Let T2 be a subtree of N. The S subtree may be rewritten into an equivalent AST subtree in which:

- the root node is labeled "function_call"
- the first child of the root node is the T2 subtree
- the second child node is labeled "parameters" and has two children: the first is the subtree T1 and the second is T3

The above rule may be written in short as:

reduce(T1, build(T2), T3) => function_call(T2, parameters(T1, T3))

The next rule represents application of the lambda-function node to the arguments in an execution plan's AST. It represents the β-reduction – a basic operation of the lambda calculus.

Rule 4.4 β-reduction for AST.

Let Q be an AST representing an execution plan and S be a subtree such that its root node R is labeled "function_call" having two child nodes: N1 labeled "lambda" and N2 labeled "parameters". Let the node L1 and subtree L2 be the left and the right child of the N1 node; subtrees T1, …, Tn be children of the N2 node and p1, …, pm be the labels of the child nodes of L1. Depending on the numbers n and m the following cases may occur:

- if m<n the syntax error should be thrown
- if m=n the tree S may be rewritten into an equivalent tree constructed out of L2 tree in which every node labeled pi has been replaced with a copy of a tree Ti, i=1...n
- if m>n the tree S may be rewritten into an equivalent tree by replacing every node labeled pi within the L2 subtree with a tree Ti, removal of all children of the N2 node and removal of N1's child nodes labeled pi, i=1...n,

The above rule and Rule 4.5 are the two main AST simplification rules. The rule 3.5 addresses the problem of the Environment Stack, crucial element of the Stack Based Approach. This rule takes its name from two fundamental operations on the ENVS.

Rule 4.5. Nested/pop elimination

Let Q be an AST representing an execution plan and T be a node labeled "tuple" having three child subtrees: T1, T2 and T3, where T1's root node is labeled "nested" and T3 is a node labeled "pop". Let x be a label of T1's leaf node. If the following conditions are met:

- the T2 subtree contains a node W labeled "nested" that has a child node labeled x
- all other nodes labeled "nested" of the T2 have bigger depth than W
- the parent node R of the W node is labeled "tuple"
- the parent of the R node is a root node for a subtree S

then the subtree S may be replaced by the R node's second child subtree.

Before we proceed with the optimization according to the described rules, we need to first check, if a query is susceptible to deforestation. To achieve this goal the optimizer should use the method of labeling the basic AST of a query with stack size and section numbers. This is exactly the same method that is used to search and factor out independent subqueries for SBQL queries. The proper algorithm was described in chapter 3.1.3 of this thesis, and its detailed description may be found in [Plod00, Subi04]. In general, when each name in a query is bound at the top most section of the ENVS, then such query does not contain independent subqueries. However, in such case the optimizer should check if this query contains a composition of at least two functions that have build/reduce definitions. If so, then the deforestation algorithm may be applied.

To start optimizing a query it needs to be rewritten into its execution plan using proper definitions presented in Table 4.1. Then an abstract syntax tree (AST) should be generated out of this plan. Next the execution plan tree should be analyzed for possible application of the reduce/build rule. This process should be performed according to Rule 4.3. Next the AST should be simplified using Rules 4.4 and 4.5. Each of those three rules should be applied as many times as possible. When none of them can be applied to the transformed execution plan tree the Rule 4.2 should be applied followed by applications of Rule 4.4 and, if possible, Rule 4.5. If no further transformation is possible, the algorithm stops its operation.

To explain how this algorithm operates let us consider an example query:

(Emp where sname = "Smith").dept (1)

Figure 4.1 presents its basic syntax tree with proper labels. The root operator is the selection (dot) operator. We may assume that the initial size of ENVS is 1 (regardless of the actual size). This assumption does not, in any way, influence the deforestation algorithm. The left child-tree of the root node would be evaluated using unchanged ENVS. However, to evaluate the right child node, a new section should be placed on the ENVS according to the evaluation rules. Thus the label *2* under the root node. The *dept* name would be bound on the second section of the ENVS, whose size would be then also equal to *2*. Other labels are placed in the same manner.

Immediate observation after all the labels have been assigned is that there are no independent subqueries. Therefore the deforestation algorithm may be applied. For the sake of shorter and clearer notation we will write *P* instead of the predicate (*sname = "Smith"*). Evaluation of this predicate is irrelevant to the deforestation technique.

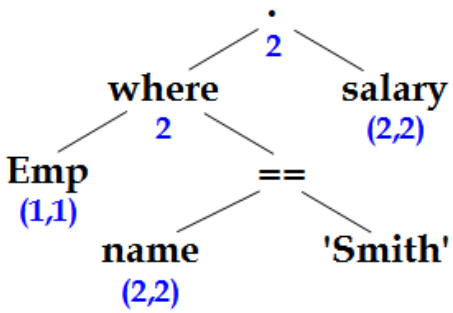


Figure 4.1: Labeled basic AST of the query (1)

The first stage of the algorithm is translating the query (1) into the composition of the basic execution plans, transforming this composition into a corresponding execution plan tree and identifying a subtree matching the requirements of the Rule 4.3. Corresponding Python language notation of the execution plan would be as follows:

```
build( lambda c,n: reduce((lambda ys,y: (nested(y), \
     reduce(c,evaluate('dept'),ys),pop())[1]), build( \
          lambda c2,n2: reduce((lambda zs,z: (nested(z), \
               (evaluate(P) and c2(zs,z) or zs),pop())[1]), \
                    evaluate('Emp'),n2)),n))
```

The inner *reduce* and *build* functions (highlighted in bold) match the requirements for application of the reduce/build rule. The same execution plan in an AST form is presented by Figure 4.2

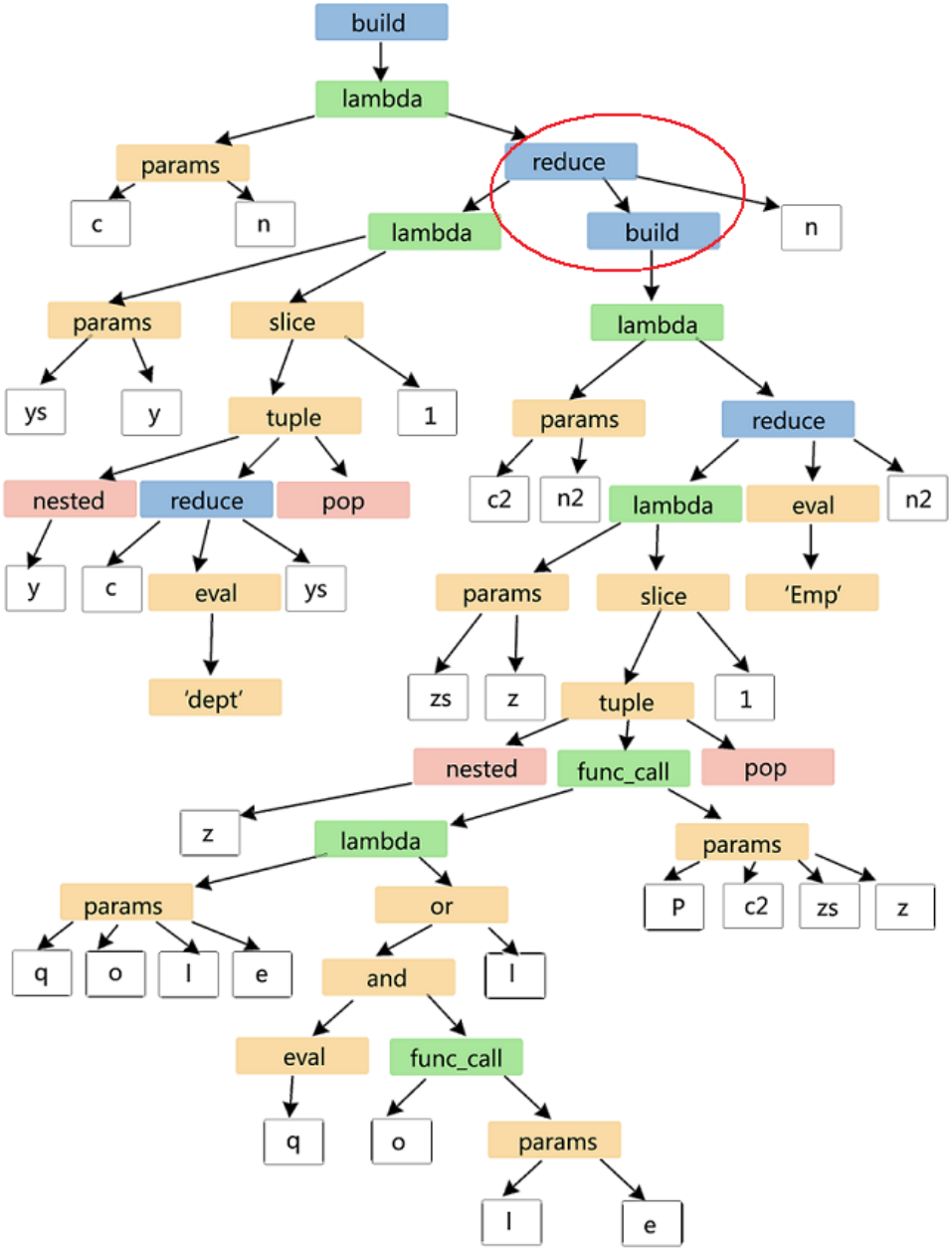


Figure 4.2: Basic execution plan tree for the query (1). The place for application of Rule 4.3 has been marked.

Figure 4.3 presents a new execution plan which was created by applying Rule 4.3 to the above tree. Additional marking has been placed to indicate where the rule 4.4 can be applied.

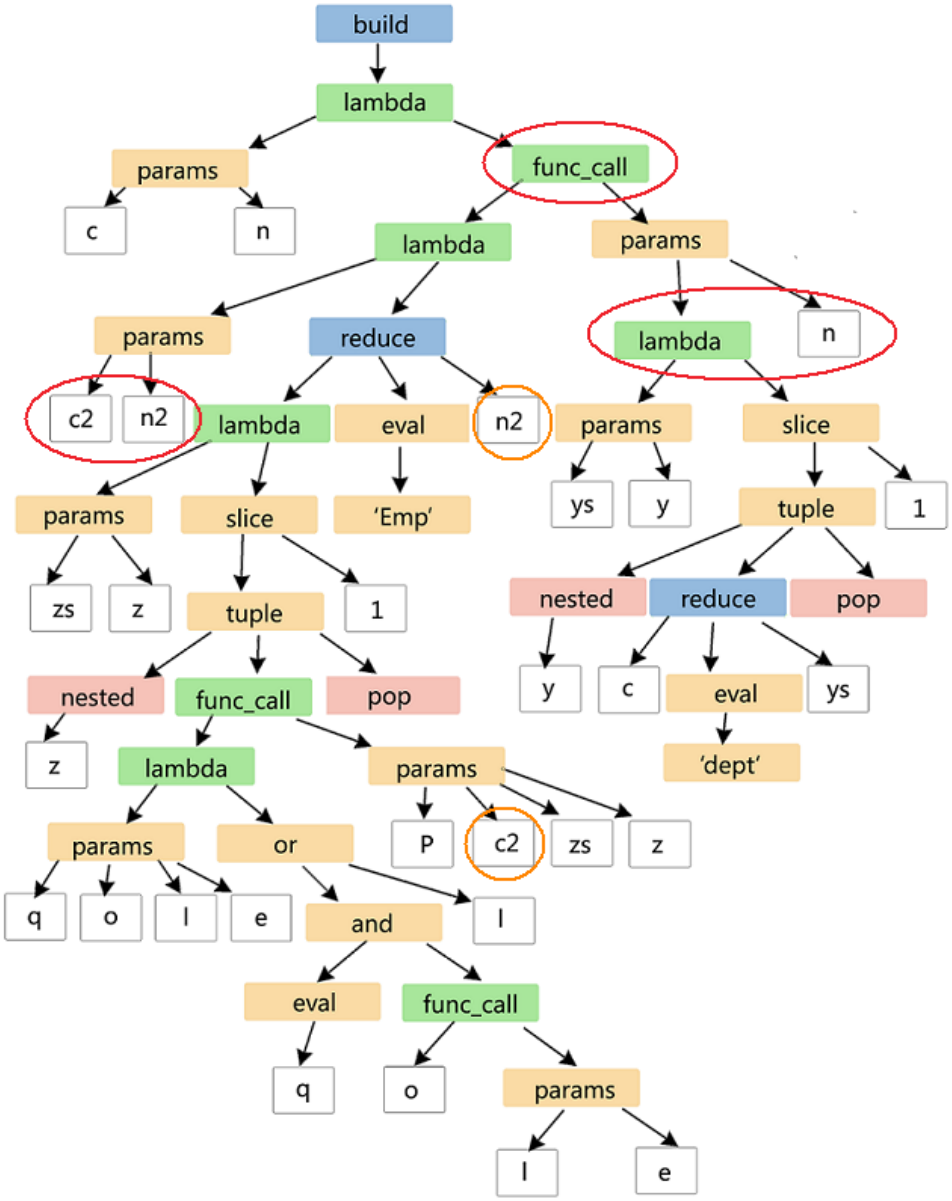


Figure 4.3: Second step of optimization algorithm for query (1)

Figure 4.4 presents the third step of an algorithm – the AST tree acquired from second step with marked places for repeated application of Rule 4.4

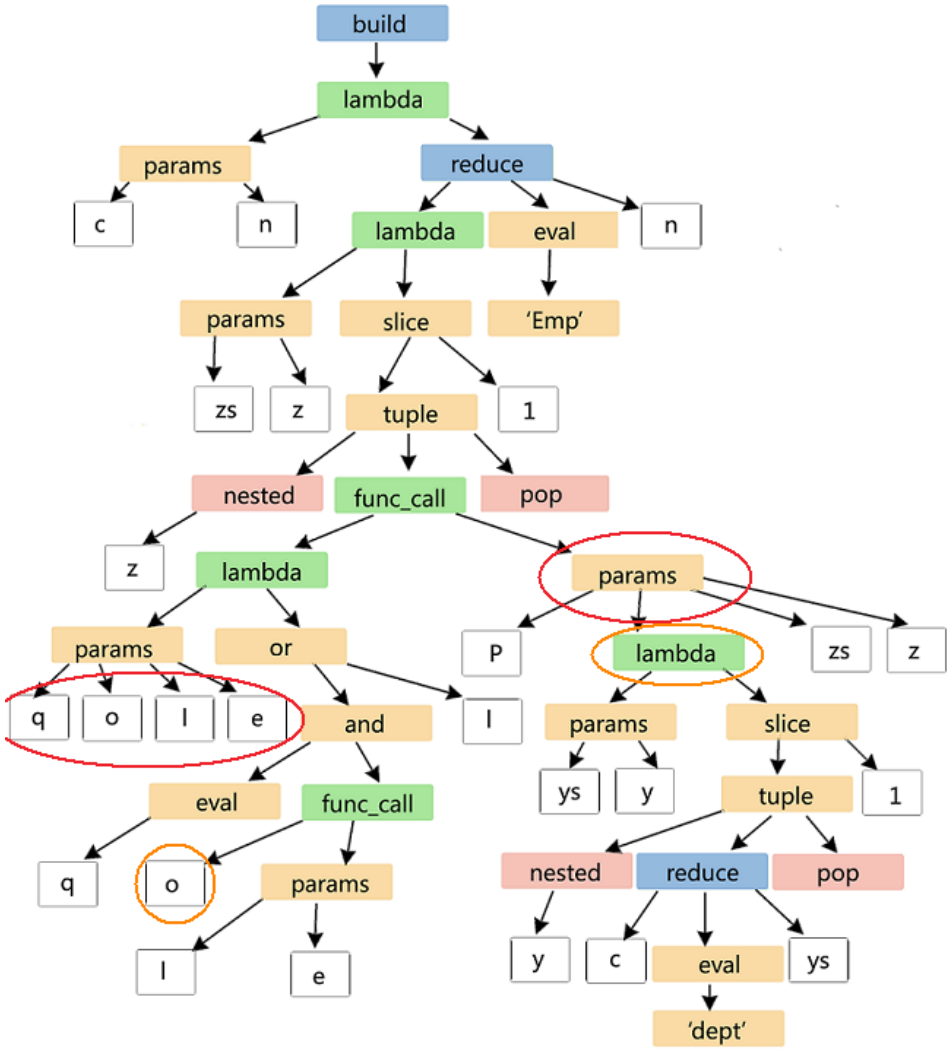


Figure 4.4: Third step of optimization algorithm for query (1)

The next two steps involve once again applying Rule 4.4 and checking for application of Rule 4.5 – the nested/pop elimination. The nodes on an AST tree subjectable to this rule have been marked on a Figure 4.6

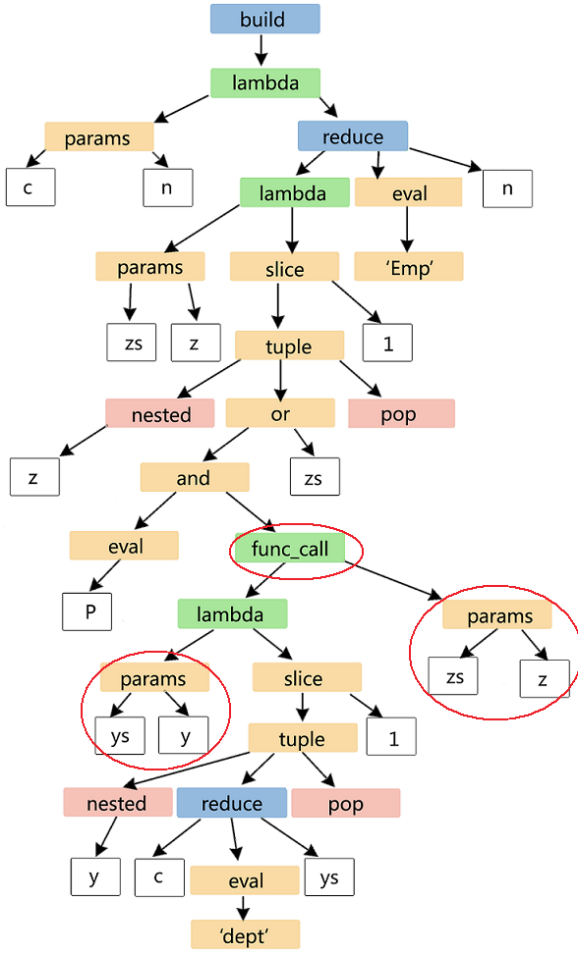


Figure 4.5: Fourth step marked for Rule 3.4

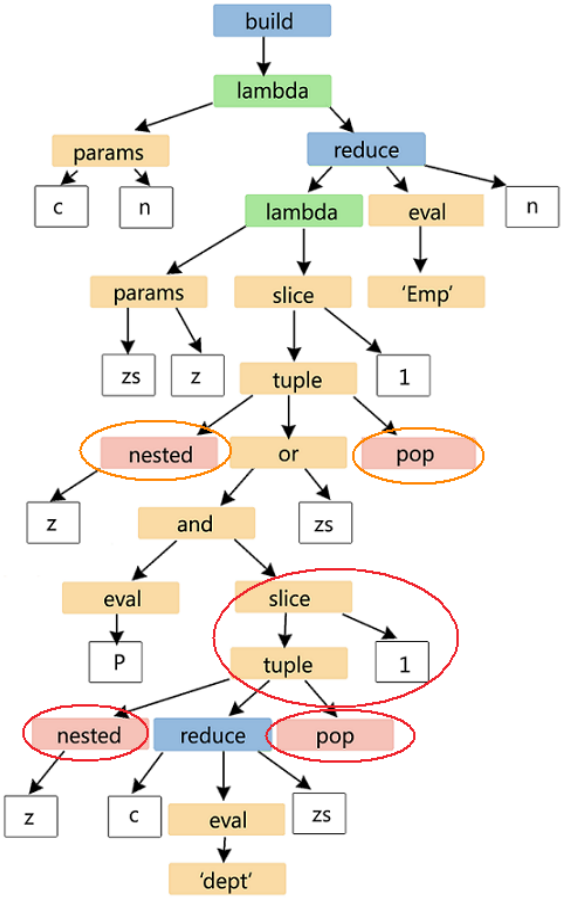


Figure 4.6: Fifth step marked for the nested/pop elimination rule

The AST tree from Figure 4.6 corresponds to the following Python/PySBQL code:

```
build( lambda c,n:reduce((lambda zs,z: (nested(z), \
     (evaluate(P) and (nested(z),reduce(c,evaluate('dept'), \
          zs),pop())[1] or zs ,pop())[1]), \
               evaluate('Emp'),n))
```

After application of the nested/pop elimination rule we acquire the following code corresponding to the tree from Figure 4.7:

```
build( lambda c,n:reduce((lambda zs,z: (nested(z), \
     (evaluate(P) and reduce(c,evaluate('dept'),zs) \
          or zs ,pop())[1]), evaluate('Emp'),n))
```

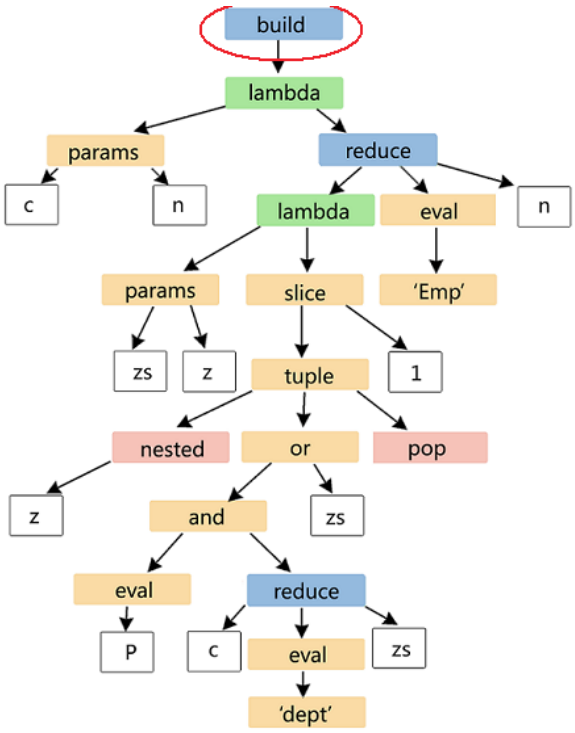


Figure 4.7: Sixth step marked for Rule 3.2

Because we cannot apply neither reduce/build transformation, nested/pop elimination nor β-reduction we now have to apply the definition of the build function according to the Rule 4.2. As a result we acquire a tree (Figure 4.8) that can be subjected to Rule 4.4

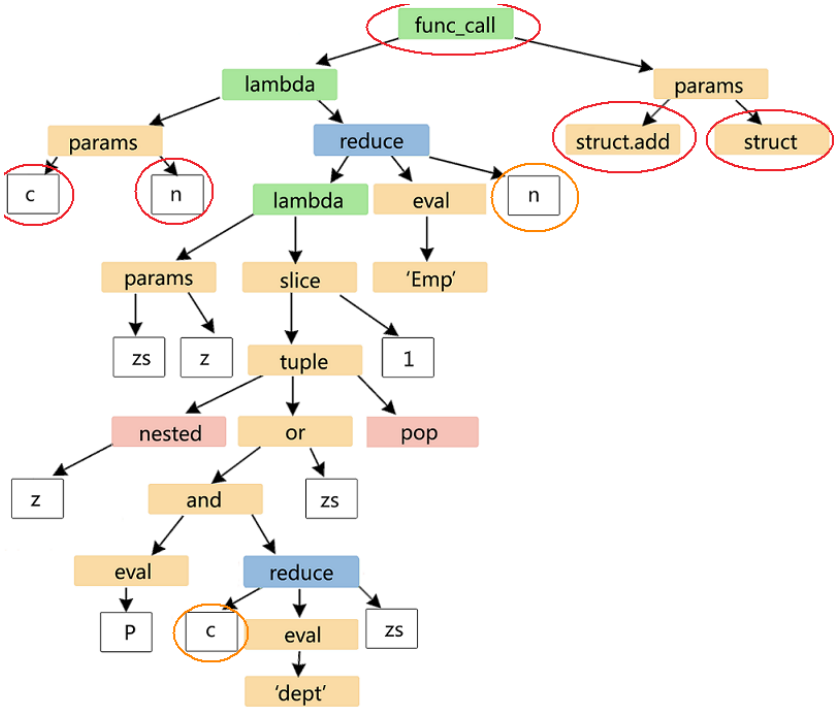


Figure 4.8: Final step of the optimization algorithm

The result of the algorithm is presented by the Figure 4.9 and the source code below:

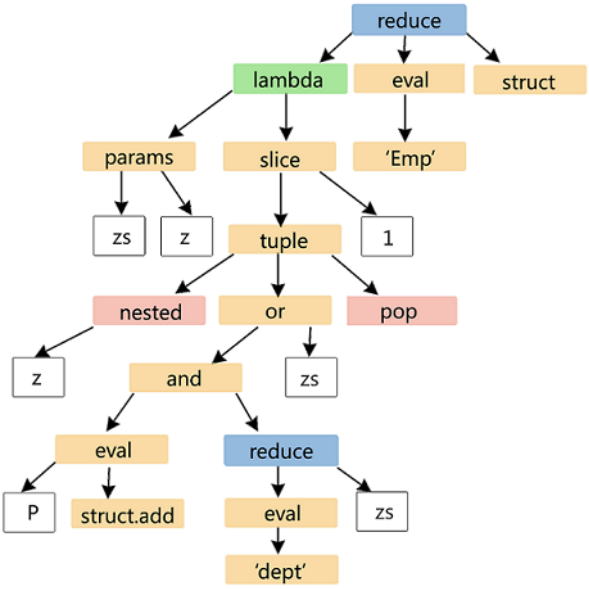


Figure 4.9: The output evaluation plan tree

```
reduce((lambda zs,z: (nested(z), (evaluate(P) and  \
     reduce(struct.__add__,evaluate('dept'),zs) or zs , \
          pop())[1]), evaluate('Emp'), struct())) \
```

During the evaluation of the not-optimized input plan one intermediate list would be created - a list of employees fulfilling the predicate *P*. In the deforested version this intermediate structure is not being created. The output plan has the following meaning: during its execution for each employee check if the surname condition is met, and if so, add their department reference to the result collection. Each employee is considered only once, what reduces resources consumption. Another benefit of this method is that the evaluation of the output program is at least as fast as evaluation of the input program in the worst case scenario, and in a better one - can speed up the process. Additionally, after a new plan for a specific composition of two operators has been generated it can be stored for future usage.

4.1.1 Efficiency Tests

The efficiency tests of the presented deforestation algorithm have been performed on PySBQL platform with XML storage. There were three data sets describing employees hierarchy: comprising 10^3, 10^4 and 10^5 employees. The tests have been performed on two machines:

- machine A with Intel core 2 duo T6400 processor, 4GB RAM memory and Windows 7
- machine B with 936X4 Athlon processor (4 cores) and 8GB RAM memory with Windows 2008 Server

The efficiency tests have been performed on the following queries:

```
(Q1) (Emp where (sname=='River' and fname=='Judy')).
        worksIn.address
(Q2) sum((Emp where (sname=='Smith' and fname=='Jane')).
        mgr.worksIn.employs)
```

On each data set and each machine the above queries gave similar results. For the set of 1000 records the original and modified queries achieved basically the same performance statistics. The average memory consumption during queries execution was 150kB. The results were returned instantly. Tables 4.1 and 4.2 present efficiency tests for queries Q1 and Q2 respectively. Columns "Memory" present amount of RAM memory consumption used during query evaluation, while "Time" shows how much time was needed for the query to return its result.

Test suite		10^3 records		10^4 records		10^5 records	
		Memory	Time	Memory	Time	Memory	Time
Original query	Machine A	150kB	<1ms	1.5 MB	6s	5.7 MB	27s
	Machine B		<1ms		1ms		2ms
Optimized query	Machine A	150kB	<1ms	0.8 MB	5s	4.6 MB	24s
	Machine B		<1ms		<1ms		1ms

Table 4.1: Results of efficiency tests for deforestation algorithm (query Q1)

Test suite		10^3 records		10^4 records		10^5 records	
		Memory	Time	Memory	Time	Memory	Time
Original query	Machine A	150kB	<1ms	1.6 MB	6s	6.4 MB	32.2s
	Machine B		<1ms		1ms		5ms
Optimized query	Machine A	150kB	<1ms	1 MB	5s	5.3 MB	29.6s
	Machine B		<1ms		<1ms		3ms

Table 4.2: Results of efficiency tests for deforestation algorithm (query Q2)

The memory consumption for both machines was almost the same; the differences were only visible in execution time. This optimization technique has shown little time improvement (by approximately 10%) and a considerable memory consumption reduction (by approximately 25%). Another conclusion coming from performed tests was that it might be profitable to implement a more efficient reduce operator that would work specifically with semi-structured data.

An additional benefit of including this method in the optimizer is that in most of the cases when deforestation cannot be applied, the optimizer can use the method of factoring out independent subqueries.

4.2 Distributivity of Algebraic Functions Over the Dot Operator

Chapter 4.1 discussed the algorithm for reduction of sizes of the intermediate structures for SBQL language. That algorithm worked on a level of execution plans based on lambda expressions. It assumed that every operator has such execution plan. For some SBQL implementations this assumption may be too troublesome and restricting. This may especially concern distributed object databases which have special execution plans that include parallel computation. This chapter presents another rewriting algorithm for reduction of sizes of the intermediate structures. The basic assumption idea of this algorithm is to rewrite an SBQL query to another SBQL query.

Let us consider a simple query:

sum(Dept.employs.Emp.salary) (3)

This query calculates the sum of salaries of all employees assuming that they are grouped by their departments. Using two techniques previously described this query may be folded into a simple function that traverses the tree of employment and adds up every encountered salary. It would require a single aggregate object. But what if we would like to make the process parallel? Or what if the database scheme is distributed and the data is fragmented? Deforestation will reduce the total amount of data transferred, but at the same time it may generate much traffic with requests addressed to distributed servers. Benefiting from the unique property of SBQL dot operator we have developed a new technique of deforestation by distributivity of linear algebraic functions over the dot operator. Our algorithm takes as an input a simple algebraic function like *sum, min, max* that has a dot expression as an argument. On output it generates a query in which after each occurrence of the navigation operator the initial function is inserted. Let us consider once again the (3) query. After modifying it with our algorithm it takes the form:

sum(Dept.sum(employs.sum(Emp.sum(salary)))) (4)

Now let us assume that in our hypothetical company we have 100 departments, each one employing at least 1000 employees. Without any optimization we have to store more than 100 000 *salary* objects in an intermediate structure. Also, most of those objects might require transfer through the network. But if we distribute the sum function, then we reduce the size of the biggest intermediate structure about 100 times. Additional profit of this method is that most of the distributed database servers may perform partial evaluation of this query, what would significantly decrease the transfer over network because instead of sending 100 *salary* objects only one number would be sent to the main server. On a distributed system where each department with its employees is stored on a different computer, the distribution of a sum function would have the biggest impact on the efficiency of the query execution.

The functions that can be distributed over the dot operator include *sum, min, max*. The *count* function might seem troublesome for the use of this technique. But when we translate *count(query)* into its equivalent *sum(query.1)* it becomes apparent that the *count* function may also be distributed. An example of optimization process for *count* function is presented below:

count((Dept where name == "IT").employs.Emp) ≡

sum((Dept where name == "IT").sum(employs.sum(Emp.1)))

Another operator that is often used in database queries is *avg* (arithmetic average) operator. This operator, like *count*, cannot be distributed over the dot operator in its basic form. But

it can be translated into a composition of a few functions (we skip here the question of implementation of execution plans):

```
avg_p(x) = ( sum(x), count(x) )

avg_p_sum(plist) = ( sum(first(plist)),
sum(second(plist)))

avg_div(x,y) = if y!=0: x/y else: 0
```

The avg_p function simultaneously increases the aggregate and the count variable. The avg_p_sum function takes a list of pairs of numbers, and returns a pair of numbers that represent the sum of respectively the first and the second elements of pairs. This function is susceptible to distributivity over the dot operator. Having those three functions we now can present a new definition of the *avg* function:

```
avg(x) = avg_div(avg_p_sum(avg_p(x)))
```

On flat collections this transformation creates intermediate structures and it slows down the evaluation. But it is meant to deal with complex path (dot) queries, and for them it has the advantage of reducing the intermediate structures and increasing the speed of evaluation. Let us consider the query:

avg((Emp where position == "Manager").subordinate.Emp.salary)

Let us assume that each of the managers has 1000 subordinates, and there are 100 managers. An intermediate structure of 100 000 database objects would be created in order to calculate the result. Now let us consider the alternative definition already in a distributed form:

avg_div(avg_p_sum((Emp where position == "Manager").

avg_p_sum(subordinate.avg_p_sum(Emp.avg_p(salary))))) (5)

During the evaluation of this query the outermost *avg_p_sum* reads from the database those employees that match the filtering condition. For each one of them it evaluates the inner *avg_p_sum* that would bind the name *subordinate* within the context of a current employee, and so on. This way the biggest intermediate structure will consist of 100 pair of numbers, which is a considerable storage saving.

4.2.1 Efficiency Tests

The testing platform was exactly the same as in the chapter 4.1.1. There were two machines – with Windows and Linux operating systems equipped with PySBQL working with an XML storage. Also, all of the data sets were the same as previously. The efficiency tests have been performed on the following queries:

```
(Q3) count(Emp.worksIn.address.('Works in ' + city))

(Q4) sum((Emp where (sname=='Smith' and fname=='Jane')).
        mgr.worksIn.employs)
```

As in the test case from the chapter 4.1.1, on each of the data sets and each machine the above queries gave similar results. Tables 4.3 and 4.4 present efficiency tests for queries Q3 and Q4 respectively. Columns "Memory" present amount of RAM memory consumption used during query evaluation, while "Time" shows how much time was needed for the query to return its result.

Test suite		10^3 records		10^4 records		10^5 records	
		Memory	Time	Memory	Time	Memory	Time
Original query	Machine A	0.1 MB	1ms	1.4 MB	5s	6.1 MB	30s
	Machine B		0ms		2ms		4ms
Optimized query	Machine A	17 kB	1ms	320 kB	1s	2.5 MB	12s
	Machine B		0ms		1ms		3ms

Table 4.3: Results of efficiency tests for query Q3

Test suite		10^3 records		10^4 records		10^5 records	
		Memory	Time	Memory	Time	Memory	Time
Original query	Machine A	160 kB	2ms	1.6 MB	6s	6.4 MB	32.2s
	Machine B		0ms		1ms		5ms
Optimized query	Machine A	57 kB	1ms	0.5 MB	1.4s	2.8 MB	15.6s
	Machine B		0ms		1ms		3ms

Table 4.4: Results of efficiency tests for query Q4

The execution times for optimized queries for the bigger tests sets show improvement of execution speed by approximately 50% for machine A and 10% for machine B. This is related to the use of memory caching on a hard drive. However, the main goal was to reduce the memory consumption. Tests have shown that for each query and each data set, the memory consumption for the optimized query was approximately 30% of the consumption for the original query.

4.3 Summary

This chapter has presented novel applications of the deforestation – an optimization technique for functional languages intended to reduce the size of intermediate structures. We have analyzed two propositions of optimization algorithms for the SBQL language. The first algorithm worked with execution plans written using Python language notation. Further research on this topic should involve generating similar algorithms for other semi-structured query languages like XQuery. The second optimization algorithm discussed in this chapter exploits functionality of SBQL's dot operator. Deforestation technique that optimizes execution plans is a stronger method than the distributivity over the dot operator, and when used together, distributivity will be overwritten. But it is not predetermined which method is better. Depending on the context and the cost model one may be preferable or more efficient than the other.

5 Optimization of Recursive Queries

In this chapter we focus on optimization of recursive queries in SQL and SBQL. Both presented techniques are based on the idea of reduction of intermediate structures. The first technique was developed for the SBQL's *close by* operator. The second algorithm was designed for SQL's recursive common table expressions, however it may also be used to optimize queries which involve recursive views.

5.1 Optimization of Recursive Queries for SBQL

The chapter 4 described two propositions of optimization techniques for SBQL queries. Those techniques are based on the shortcut fusion technique described in the chapter 3.2.

Although shortcut deforestation for SBQL queries is very efficient in eliminating intermediate structures, it has a big drawback — it does not optimize recursive calls. The close by operator representing transitive closure of the dot operator cannot be translated into *reduce/build* notation, thus it falls outside of the cheap deforestation technique. The need for optimization of recursive SBQL queries resulted in the research and development of a new algorithm that was based on rewriting of execution plans. Preliminary results of this research have been published in [Burz10]. The algorithm introduced in that paper is used to eliminate intermediate structures that are generated during evaluation of a composition of a *close by* operator and an aggregate function. The construction of this algorithm has been inspired by lightweight fusion technique for functional languages described in [Ohor07].

Before describing the above mentioned algorithm we first need to introduce an execution plan for the *close by* operator written using Python language notation.

Definition 5.1.

The execution plan for the *close by* operator is represented with the following recursive function definition and call:

```
def closeby (dotFunction, queryRes):
     if isEmpty(queryRes): return bag()
     else:
       return bag.__add__(queryRes,
                closeby (dotFunction, dotFunction(queryRes))
closeby(makeDotF(leftQuery),eval(rightQuery))
```

Where `dotFunction` represents the execution of the left query in context of the result bag of the right query.

Our algorithm is composed out of three steps:

Method 5.1

Let Q be a close by query and A be an aggregate function that takes Q as an argument. In order to eliminate intermediate structures created by Q the following steps should be undertaken:

- inline the A function's call into both *return* clauses of the *close by*'s execution plan function
- simplify all calculation that can be computed without searching through the database section by performing built-in operations such as adding numbers or processing strings
- generate a new execution plan function representing the composition of the analyzed operators and replace the A(Q) call with this execution plan.

The above rule utilizes only elementary operations used also during normal evaluation process. What is important in this rule is the order of the steps necessary and the simplification stage.

The newly generated execution plan function may be stored for the commonly used compositions. Let us analyze this algorithm on a composition of a *count* function with *close by* operator. A sample definition of the *count* function is presented below:

Definition 5.2. Definition of the count function

```
def count(list_arg):
     if isEmpty(list_arg): return 0
     else:
          len = 0
          for i in list_arg:
               len+=1
          return len
```

According to the Rule 5.1 we firstly inline the *count* function into the definition of the *close by*'s execution plan function:

```
     if isEmpty(queryRes): return count(bag())
     else:
       return count(bag.__add__(queryRes,
              closeby (dotFunction, dotFunction(queryRes)))
```

Now we simplify the calculation acquiring:

```
     if isEmpty(queryRes): return 0
     else:
       return count(queryRes) +
          count(closeby (dotFunction, dotFunction(queryRes)))
```

The last step is to generate a new execution plan function and use it in place of *count(query1 closeby query2)* call:

```
def count_closeby (dotFunction, queryRes):
     if isEmpty(queryRes): return 0
     else:
```

```
        return count(queryRes) +
           count_closeby (dotFunction, dotFunction(queryRes))
count_closeby(makeDotF(leftQuery),eval(rightQuery))
```

This new function will calculate the same result as the initial query, but it does not use an intermediate structure containing all of the database elements that are retrieved during the evaluation. Instead it counts those elements at each iteration of the recursion. For multilevel hierarchy it significantly reduces the size of intermediate structures, because the maximum size of such structure is equal to the sum of objects acquired in a given iteration. This technique can be efficiently combined with reduce/build rules described in the chapter 4.

5.1.1 Efficiency Tests

Similar to the testing of algorithms from the fourth chapter, the efficiency tests of the presented algorithm have been performed on PySBQL platform with XML storage. There were three data sets describing employees hierarchy: comprising 10^3, 10^4 and 10^5 employees. The tests have been performed on two machines:

- machine A with Intel core 2 duo T6400 processor, 4GB RAM memory and Windows 7
- machine B with 936X4 Athlon processor (4 cores) and 8GB RAM memory with Windows 2008 Server

The efficiency tests have been performed on the following query:

```
sum(((Emp where (sname=='Smith' and fname=='Jane'))
        close by mgr).salary)
```

On each data set both machines have shown similar memory consumption. The data has been prepared so that it did not contain cycles. Table 5.1 presents efficiency tests for the above query. Columns "Mem" present amount of RAM memory consumption used during query evaluation, "Time A" shows how much time was needed for the query to return its result on machine A, while "Time B" - on machine B.

Test suite	10^3 records			10^4 records			10^5 records		
	Mem	Time A	Time B	Mem	Time A	Time B	Mem	Time A	Time B
Original query	0.9 MB	1.3s	84ms	12.6 MB	16s	2.1s	47.3 MB	368s	39s
Optimized query	0.8MB	1s	68ms	10.9 MB	14s	1.5s	38.6 MB	307s	32s

Table 5.1: Results of efficiency tests of close by operator optimization

This optimization technique achieved approximate 14% reduction of memory consumption. The observed difference in execution speed for the biggest test suite for machine A is related to the heavy use of memory caching on a hard drive. Tests performed using different "starting" elements have shown similar results.

This tests show that the presented technique of query rewriting is efficient in optimizing SBQL's recursive queries. Additional research around this type of SBQL's queries could be based on extending this algorithm with a heavy use of the meta-data and indexing.

5.2 Pushing Predicates into Recursive SQL Common Table Expressions

Queries based on the recursive Common Table Expressions can be found in most of the popular Database Management Systems. Such queries are very troublesome because of their complexity, resource consumption and possibility of existence of endless loops. It is also very difficult to find an efficient execution plan for them. The work [Przy10] presents the results of efficiency test for recursive queries' implementations. It shows that even for small amounts of data, the evaluation time could be very big.

The evaluation time and huge amounts of intermediate data created during recursive query's evaluation lead to development of an algorithm for the optimization of recursive CTEs through rewriting – thus leaving a possibility for the usage of other optimization algorithms. This method has been described in [Burz09]. It was inspired by the method of predicate-move-around described

in chapter 3 of this thesis. However, this method applies to non-recursive queries only. Recursive queries are much more complex, since predicates external to them apply to the nodes obtained during the execution. It could be useful to push such predicates into the initial step or the recursive step. We cannot do it straightforwardly, since the predicate holding for the resulting nodes does not have to hold for neither intermediate recursive results nor the initial recursion step. The new method of pushing predicated into CTE is subtle enough not to change the semantics of the query.

Together with pushing predicates this method also tries to push other operators into the recursive CTE, so that some part of computation would be performed on the fly together with the recursive processing. This spares space needed for temporary data structures and the time needed to store and retrieve data from them. This part of the optimization method is inspired by the deforestation developed for functional languages.

5.2.1 Motivating Example

The best way to present the idea behind this algorithm is to show its potential applications. Let us consider a database table *Emp* with four columns: *eid*, *ename*, *salary* and *mgr*. The column *eid* is the primary key, while *mgr* is a foreign key which references *eid*. The column *mgr* stores data on managers of individual employees. Top managers have NULL in this column. We define also a recursive view which shows the subordinate-manager transitive relationship, i.e. it prints pairs of *eids*, such that the first component of the pair is a subordinate while the second is his/her manager. Following SQL-99 standard a query expressing this structure would have the form:

```
CREATE VIEW subordinates (seid, meid) AS
     WITH RECURSIVE subs(seid, meid) AS
          SELECT e.eid AS seid, e.eid AS meid FROM Emp e
          UNION ALL
          SELECT e3.eid AS seid, s.eid AS meid
               FROM Emp e3 JOIN subs s ON (e3.mgr = s.seid)
     SELECT * FROM subs;
```

This view can then be used to find the total salary of all subordinate employees of, say, Smith:

```
SELECT SUM(e2.salary)
     FROM subordinates s2
     JOIN Emp e2 ON (e2.eid = s2.seid)
     JOIN Emp e1 ON (e1.eid = s2.meid)
     WHERE e1.ename = 'Smith';
```

A naïve execution of such a query consists in materializing the whole transitive subordinate relationship. However, we need only a small fraction of this relationship which concerns employees named Smith and their subordinates.

In order to avoid materializing the whole view, we start from a standard technique of query modification. We expand the view definition:

```
WITH RECURSIVE subs(seid, meid) AS
     SELECT e.eid AS seid, e.eid AS meid FROM Emp e
     UNION ALL
     SELECT e3.eid AS seid, s.eid AS meid
          FROM Emp e3 JOIN subs s ON (e3.mgr = s.seid)
SELECT SUM(e2.salary)
     FROM (SELECT * FROM subs) s2
     JOIN Emp e2 ON (e2.eid = s2.seid)
     JOIN Emp e1 ON (e1.eid = s2.meid)
     WHERE e1.ename = 'Smith';
```

The execution of this query can be significantly improved, if we somehow manage to push the predicate e1.ename = 'Smith' to the first part of the CTE. After this first improvement it is possible to get rid of the join with e1 and push the join with e2 as well as the retrieval of the salary

into the CTE. After all this changes we get the following form of our query:

```
WITH RECURSIVE subs(seid, salary) AS
     SELECT e.eid AS seid, e.salary
          FROM Emp e
          WHERE e.ename = 'Smith'
     UNION ALL
     SELECT e3.eid AS seid, e3.salary
          FROM Emp e3 JOIN subs s ON (e3.mgr = s.seid);
SELECT SUM(s2.salary) FROM subs s2;
```

The result of the predicate push and the query fusion is satisfactory. Now we traverse only the Smith's hierarchy. Further optimization is not possible, by rewriting SQL query to another SQL query (SQL:99 severely limits the form of recursive CTEs). However, we do not need to accumulate neither *eids* nor salaries. We just need to have one temporary structure, i.e. a number register to sum the salaries on the fly as we traverse the hierarchy. This is the most robust plan (traverse the hierarchy and accumulate salaries). Such rewriting is a simple application of deforestation and can be done by a DBMS on the level of query execution plans even if it is not expressible in SQL:99.

5.2.2 Utility Optimizations

Let us now discuss the algorithm that would accomplish previously presented rewriting of a recursive query. In general it may be divided into the following steps performed interchangeably:

- expanding the view definition with substitution of variable names
- elimination of vain joins
- elimination of self-joins on primary keys (primary key-to-primary key self-join elimination)
- predicate push-in

The first three steps are well known in the field of optimization by rewriting SQL queries. They will be briefly described in this sub-chapter including the basic assumptions that should be met for them. The last step of pushing-in predicates, being the key point of our algorithm, will be described in detail in the sub-chapter 5.2.3

The first step of our algorithm is purely syntactic and performed only once. The algorithm begins by expanding the recursive view's definition. The immediate step should be so called α-conversion – basically substitution of the variable names.

Rule 5.2. Alpha-conversion for Predicate Push-In

Let Q be a recursive query using CTE and containing references to tables T_1,..., T_n, where T_k,..,T_n are tables that have not been given alias names, 1<=k<=n. In order to acquire an equivalent query with respect to alias names the following step should be undertaken:

a. starting from table T_1 up to T_{k-1} the given table alias should be replaced with new, unique alias name and corresponding column calls should be renamed accordingly

b. tables T_k,...,T_n should be assigned new unique alias names and column calls corresponding to those tables should be renamed to include those alias names

c. column names included in the CTE's header should be replaced with new, unique alias names. Those alias names should be assigned to corresponding column definitions from the inner SELECT clauses. The references to those columns should be renamed accordingly

This Rule is used to introduce order in the alias names and definitions. This is done to avoid potential problems in the further stages of the main algorithm.

The second technique is the elimination of vain joins. By vain join we understand a predicate joining two tables based on primary key-foreign key dependencies where the table joined by its primary key is not used in any other clause or predicate of the given subquery. The technique of vain join elimination is usually applied after some other query transformation.

Rule 5.3. Removal of vain joins from the CTE and outer query

> Let Q be a recursive query using CTE. Let us use the names T_1,T_2 for the tables with alias names TA_1, TA_2 respectively. If T_1's primary key is used in a joining condition with a foreign key of table T_2, but besides this joining condition it is not used in any other way. If the foreign key column of T_2 table does not contain NULL values then this joining condition and the reference to the table T_1 may be removed from the query Q without changing the result of Q. If the foreign key column of T_2 table contains nulls then the join predicate should be replaced with the *T_2.foreign_key IS NOT NULL* predicate and the reference to the table T_1 removed from the appropriate FROM clause

The Rule 5.3 may be applied to joining condition occurring in any of the parts of the CTE, or in the outer query that uses the CTE. The subtle issue is the NOT NULL condition for the foreign key of the T_2 table. If the foreign key column would contain null values, then the joining condition would have the same functionality as IS NOT NULL condition. But if the schema determines the foreign key to be NOT NULL, this condition is useless and is not added.

Another simple conversion is a self-join elimination when the join is one-to-one (primary key to primary key).

Rule 5.4. Primary key-to-primary key self-join elimination

> Let Q be a recursive query using CTE. Let T be a table referenced inside Q under two alias names TA_1 and TA_2, such that the query Q contains a predicate joining TA_1 with TA_2 using their primary keys. The query Q may be rewritten into equivalent query by deleting the marked self-joining condition, deleting the reference to TA_2 from the FROM clause and replacing each remaining occurrence of alias name TA_2 with TA_1

This technique may be illustrated by the following example. Starting from a query:

```
WITH subs(seid, meid, salary) AS (
     SELECT e.eid AS seid, e.eid AS meid, e2.salary as salary
          FROM Emp e, Emp e2
          WHERE e.eid = e2.eid
```

```
     UNION ALL
     SELECT e3.eid AS seid, s.meid AS meid, e4.salary as salary
          FROM Emp e3, subs s, Emp e4
          WHERE (e3.mgr = s.seid) AND e.eid = e4.eid )
SELECT SUM(s2.salary)
     FROM subs s2 JOIN Emp e1 ON (e1.eid = s2.meid)
     WHERE e1.ename = 'Smith';
```

The Emp table's instances e1 and e2 are joined using their primary keys. We mark the e2 table for removal from the initial query using the Rule 5.4. As a result we obtain the query:

```
WITH subs(seid, meid, salary) AS (
     SELECT e.eid AS seid, e.eid AS meid, e.salary as salary
          FROM Emp e
     UNION ALL
     SELECT e3.eid AS seid, s.meid AS meid, e4.salary as salary
          FROM Emp e3, subs s, Emp e4
          WHERE (e3.mgr = s.seid)
          AND e.eid = e4.eid )
SELECT SUM(s2.salary)
     FROM subs s2 JOIN Emp e1 ON (e1.eid = s2.meid)
     WHERE e1.ename = 'Smith';
```

Primary key-to-primary key self-join elimination may be applied to both parts of a CTE definition and to the main part of the query. In the mentioned example it was applied to the first part of the CTE, but it might have been also applied to the recursive step of that query.

5.2.3 Predicate Push into Common Table Expressions

This section describes the main part of our technique, i.e. how to find predicates which can be pushed into a CTE and how to rewrite the query to push selected predicates into CTE. In subsequent steps we will analyze each table (represented by some alias name) joined to the result of a CTE. Such a table may be simply used in the query surrounding the CTE or for example may appear to be joined with CTE after expansion of the definition of a view (as in the example from Section 5.2.1). In the following paragraphs we will call such a table as "marked for analysis".

Depending on the part of the analyzed query where the marked table's alias is called we have three transformations:

Rule 5.5. Join predicate pushing

Let Q be a recursive query using CTE and T be a table with alias name TA such that TA is marked for analysis and Q contains a predicate joining TA with CTE. In order to push the joining predicate into the CTE, the following steps should be undertaken:

- copy the table T's declaration into all of the inner FROM clauses
- copy the joining condition into the WHERE clauses of the CTE translating CTE's column call into its equivalent within the part of the CTE being processed.

The first part of the Rule 5.5 is fairly intuitive. As for the second part the action that might be unclear is translation of the CTE's column call used for joining into its equivalent. Let us analyze an example of how this action might be performed.

Let us assume that CTE's column used in the joining predicate has been named $cte.C_1$. In the first SELECT clause of the CTE we search for an alias name definition for C_1. The Rule 5.2. (Alpha-conversion) guarantees the existence of such definition. When we find $TA_i.C_j$ *AS* C_1 we substitute the column name $cte.C_1$ with $TA_i.C_j$. We proceed analogously when copying the join condition into the recursive part of the CTE

Rule 5.6. Selection clause extension

Let Q be a recursive query using CTE and T be a table with alias name TA such that TA is

marked for analysis. Let Q contain a predicate joining TA with CTE using table T's primary key and the Q's outer SELECT clause contains calls to columns $TA.C_1$,.., $TA.C_n$. In order to create a query Q_1 resulting in the same set of records as Q, the following steps should be undertaken:

- push-in the joining condition (using the Rule 5.5)
- copy the columns $TA.C_1$,..., $TA.C_n$ calls into all inner SELECT clauses, assigning those columns' calls new alias names (NC_1,..., NC_n accordingly)
- expand CTE's header using aliases NC_1,..., NC_n.
- in the outer SELECT clause replace the alias TA with the outer alias of the CTE.

The above rule has application to cases when a table is joined in the outer query to the CTE and is also referenced in the outer select clause. The next rule has similar construction but is applied to cases when the outer select query contains a table joined to the CTE and to some other table.

Rule 5.7. CTE extension

Let Q be a recursive query using CTE and T be a table with alias name TA such that TA is marked for analysis. Let Q contain a predicate joining TA with CTE using table T's primary key and a predicate joining TA with table R having the alias name RA, RA!=TA. Let $TA.C_1$,..., $TA.C_n$, $RA.A_1$,...,$RA.A_n$ be column calls used to join TA with RA. In order to create a query Q_1 resulting in the same set of records as Q, the following steps should be undertaken:

- push-in the predicate joining TA with CTE (using the Rule 5.5.)
- copy the column calls $TA.C_1$,..., $TA.C_n$ used in the predicate joining into all inner SELECT clauses, assigning those columns' calls new unique alias names (NC_1,..., NC_n accordingly)
- expand CTE's header using aliases NC_1,..., NC_n.
- in the outer SELECT clause replace the calls $TA.C_1$,..., $TA.C_n$ with $cte.NC_1$,...,NC_n

If a recursive query involving CTE matches the conditions for rules 5.5, 5.6 and 5.7 for the same table alias marked for analysis, those rules should be performed together. The result of applying those three rules is illustrated by the following example:

Having the query:

```
WITH subs(seid, meid) AS (
     SELECT e.eid AS seid, e.eid AS meid FROM Emp e
     UNION ALL
     SELECT e3.eid AS seid, s.meid AS meid
          FROM Emp e3, subs s
          WHERE e3.mgr = s.seid )
SELECT e2.salary, d1.name
     FROM subs s2 JOIN Emp e2 ON (e2.eid = s2.seid)
     JOIN Emp e1 ON (e1.eid = s2.meid)
     JOIN Dept d1 ON (e2.dept = d1.did)
     WHERE e1.ename = 'Smith';
```

The table marked for analysis is *Emp e2*. This table is used in two join conditions (with the CTE, and with the Dept table) and once in the SELECT clause, thus meeting the conditions for all three rules. By applying the Rule 5.5 we copy the table name into both FROM clauses existing in the CTE definition, also we copy the predicate joining *e2* with the CTE and the *e2*'s column calls *(salary* and *dept)*. While copying those calls and predicates we assign new alias names for columns and extend the CTE's header.

Finally we apply the Rule 5.3 and remove the marked table with its references from the outer selection query. The resulting query has the form:

```
WITH subs(seid, meid, dept, salary) AS (
     SELECT e.eid AS seid, e.eid AS meid,
          e2_1.dept AS dept, e2.salary AS salary
          FROM Emp e, Emp e2
          WHERE e2_1.eid = e.eid
     UNION ALL
```

```
    SELECT e3.eid AS seid, s.meid AS meid,
        e2_2.dept AS dept, e2_2.salary AS salary
        FROM Emp e3, subs s, Emp e2_2
        WHERE e3.mgr = s.seid AND e2_2.eid = e3.eid )
SELECT s2.salary, d1.name
    FROM subs s2 JOIN Emp e1 ON (e1.eid = s2.meid)
    JOIN Dept d1 ON (s2.dept = d1.did)
    WHERE e1.ename = 'Smith';
```

This form may undergo further optimizations like elimination of self-join. One thing has to be mentioned: if the marked table is not joined with CTE, is should be skipped and returned to later, after other modifications to CTE.

The last and most important case is when a table from the outer query is referenced within a predicate other than join. It should be marked for pushing into CTE, but before moving into CTE we have to check if moving this predicate into CTE is possible. There are many predicates for which pushing them into CTE would put too big restrictions on the CTE resulting in loss of data. During the research on recursive queries we found that the predicate can be pushed into the CTE only if we can isolate a sub-tree of the result tree that contains only the elements fulfilling the predicate and no other node outside this sub-tree fulfills this predicate. Let us imagine a situation, when a hierarchy tree contains some nodes matching the given predicate, but those nodes are placed randomly along the branches. Predicate pushing could result in elimination of a branch containing a matching node, which should be included in the result set.

The availability of predicate pushing may be only verified by checking for the existence of the tree invariant – an attribute of a node which value is the same for all the nodes on a given branch. So a general method for pushing a predicate into CTE is based on checking CTE for the existence of tree invariant and if found, checking if the predicate can be attached to CTE through this invariant. To perform this check we use induction rules.

Rule 5.8. Predicate pushing into recursive CTE

Let Q be a recursive query using CTE, S_I be the SELECT subquery forming initial step of Q, S_R be the SELECT subquery forming the recursive step. In order to check for the existence of the tree invariant the following steps should be performed:

- Create the schema of the initial tuples by analyzing the SELECT clauses from the SI subquery or the header of the CTE
- Form a general representation of such tuples (a_1, a_2, … a_n), where n is the length of the initial tuples
- By analyzing SR subquery (SELECT clause and join predicates) form a new tuple (b_1, b_2, …, b_n) out of the (a_1, a_2, …, a_n)
- For each 1<=i<=n compare a_i with b_i. If an equality is found mark the number *i* as the index of the tree invariant. If no equality exists, the predicates cannot be pushed in
- If a_i is found to be the tree invariant, and there exists a predicate that could be attached to this column through a join condition, such predicate may be pushed into the SI subquery.

Based on the induction rules, if a filtering condition is attached to a tree invariant, then each tuple formed from a tuple matching this condition also satisfies it. So it is sufficient to push the predicate only to the SI subquery by pushing in the appropriate joining condition (if necessary) using Rule 5.5 and moving the filtering predicate from outer query into SI.

What is important is that the filtering predicate does not need to be an equality condition. Let us now observe how this method is performed on an example. Let us analyze the following query (with the join condition already pushed in):

```
WITH subs(seid, meid, salary) AS (
     SELECT e.eid AS seid, e.eid AS meid, e.salary as salary
          FROM Emp e, Emp e1
          WHERE e1.eid = e.eid
```

```
    UNION ALL
    SELECT e3.eid AS seid, s.meid AS meid,e3.salary as salary
        FROM Emp e3, subs s, Emp e1
        WHERE e3.mgr = s.seid AND e1.eid = s.meid )
SELECT SUM(s2.salary)
    FROM subs s2 JOIN Emp e1 ON (e1.eid = s2.meid)
    WHERE e1.ename = 'Smith';
```

In the CTE definition we reference the table *Emp* four times and once the CTE itself. The table *Emp e1* occurs in the predicate *e1.ename = 'Smith'*.

By analyzing SELECT clauses of the CTE we find that the initial step generates tuples of the form:

(e, e, s_e)

Let us assume that tuple $(a, b, c) \in CTE$. Querying the meta-data gives us the information about the Emp table including the list of the attributes: (EID, ENAME, MGR, SALARY), their types and the table's primary key (in this case the EID column). This means that every tuple belonging to the relation Emp has the form:

(e, n_e, m_e, s_e).

All of the tuple's elements are functionally dependent on the first element. During the recursion step from this tuple the following tuples are generated:

$((a, b, c), (e1, f_{e1}, l_{e1}, a, s_{e1}), (b, f_b, l_b, m_b, s_b))$

Next by projection on the 4-th,2-nd, and 8-th element we form a tuple:

$(e1, b, s_{e1})$

Comparing this tuple with the initial tuple template we see that the second parameter is a tree invariant, so we may attach to this parameter a table with predicate limiting the size of the result collection. Because the predicate *e1.ename = 'Smith'* references a table that is joined to the second element of the generated tuple, this predicate can be pushed into the initial step of CTE.

By applying Rule 5.3 to the outer select query, and Rule 5.4 to both inner SELECT subqueries we acquire a query:

```
WITH subs(seid, meid, salary) AS (
     SELECT e.eid AS seid, e.eid as meid, e.salary  as salary
          FROM Emp e
          WHERE e.ename = 'Smith'
     UNION ALL
     SELECT e3.eid AS seid, s.meid as meid, e3.salary as salary
          FROM Emp e3, subs s
          WHERE e3.mgr = s.seid )
SELECT SUM(s2.salary)
     FROM subs s2;
```

This way we have obtained a query which traverses only a fraction of the whole hierarchy. It is the final query of our motivating example (see Section 5.2.1). The predicate *e1.ename = 'Smith'* has been successfully pushed into the CTE. The general procedure of optimizing recursive SQL query is to firstly push in all the predicates and columns possible and then to use simplification rules described in 5.2.2.

5.2.4 Measured Improvement

This section presents the results of tests performed on two sets of queries – the motivating example and trains' routes. The tests have been performed on two machines:

- machine A with Intel core 2 duo T6400 processor and 4GB RAM memory and Windows Vista OS and MS SQL Server 2008, PostgreSQL 8.4 and IBM DB2 9.7 databases
- machine B with 2500+ Athlon processor and 1GB RAM memory with Ubuntu 9.10 OS and PostgreSQL 8.4 and IBM DB2 9.7 databases

```
WITH RECURSIVE subs(seid, meid) AS(
     SELECT e.eid AS seid, e.eid AS meid FROM Emp e
     UNION ALL
     SELECT e3.eid AS seid, s.meid AS meid
          FROM Emp e3 JOIN subs s ON (e3.mgr = s.seid))
SELECT SUM(e2.salary)
     FROM (SELECT * FROM subs) s2
     JOIN Emp e2 ON (e2.eid = s2.seid)
     JOIN Emp e1 ON (e1.eid = s2.meid)
     WHERE e1.ename = 'Smith';
```

Query 5.1: Calculates the sum of salaries of all Smith's subordinates

The first test suite dealt with data is stored within a table *Emp(eid, ename, mgr, salary)* containing 10 000 records. The hierarchy itself was created in such a way to eliminate cycles (which is common in a company hierarchy). The query being tested is Query 5.1

The second suite of test includes two tables: Cities(cid, city) containing 200 distinct entries and Trains(Tid, departure, arrival, railname, price) containing 3000 records. The basic query being tested is Query 5.2 presented below:

```
WITH  destinations (origin, departure, arrival, connections) AS
   (SELECT a.departure, a.departure, a.arrival, 1
           FROM trains a
    UNION ALL
    SELECT r.origin, b.departure, b.arrival, r.connections + 1
           FROM destinations r, trains b
           WHERE r.arrival = b.departure
           AND r.connections < I )
SELECT count(*)
   FROM destinations e, cities c
           WHERE e.origin = c.cid
           AND c.city = 'Warsaw';
```

Query 5.2: Calculates the number of possible train routes originating from Warsaw with limitation placed on number of connections

The parameter *I* was used to limit the recursion depth. It was set to a number ranging from 0 to 5 – bigger numbers resulted in memory allocation errors during execution of not optimized queries. In addition to the basic test queries, modified starting points for both suites have been tested. However, the general ratio of evaluation time before and after optimization was the same in each case.

The table 5.2 presents the results of the efficiency tests performed on two schemes: corporate hierarchy and train connections. The minus sign indicates that the DBMS returned a memory allocation error.

Test suite		I	Machine A (Windows)			Machine B (Linux)	
			SQL Server	PostgreSQL	DB2	PostgreSQL	DB2
Original query	Subordinates	-	1,47s	132ms	50,1s	235ms	79,36s
	Trains	0	30 ms	4ms	6ms	15ms	5ms
		1	46 ms	4ms	6ms	15ms	5ms
		2	0,7 s	68ms	0,82s	186ms	1,25s
		3	6,78 s	0,68s	9,53s	1,02s	14,92s
		4	53s	5,04s	87,92s	11,82s	136,43s
		5	348,76s	-	673,15s	-	-
Optimized query	Subordinates	-	93ms	26ms	81ms	60ms	124ms
	Trains	0	0ms	3ms	1ms	2ms	2ms
		1	16ms	3ms	1ms	2ms	2ms
		2	46ms	7ms	22ms	7ms	59ms
		3	0,47s	76ms	0,52s	83ms	1,42s
		4	5,16s	0,5s	6,95s	0,83s	20,58s
		5	45,98s	4,87s	73,75s	16,94s	223,47s

Table 5.2: Results of efficiency tests

The execution times for optimized queries are approximately 10 times better when the depth of recursion exceeds 3. For the recursion depth lower than 3 because of the time expenditure needed to optimize the query the execution times would be similar. The execution plans (Figures from 5.1 to 5.4) of all four types of queries for IBM DB2 database provide some insight into the differences in execution time for bigger amount of recursion steps.

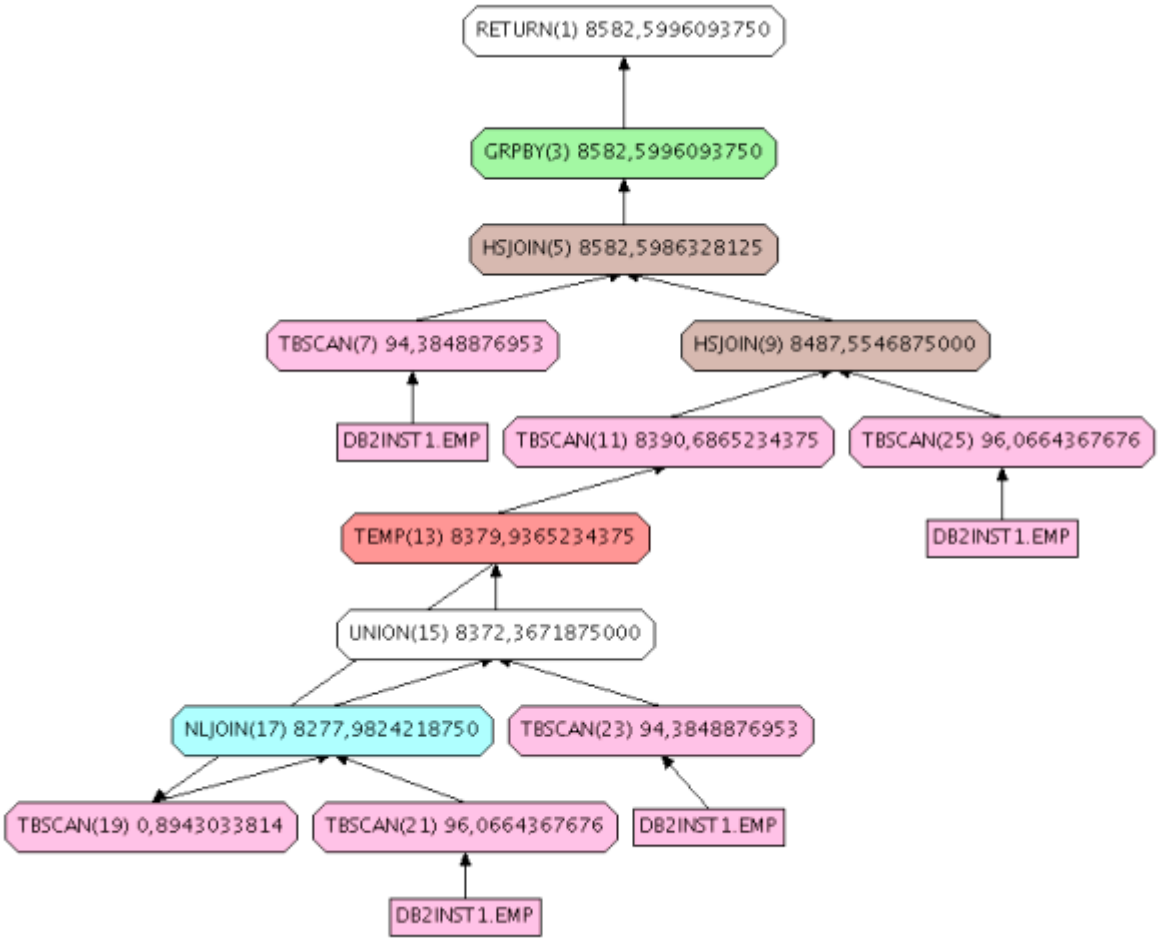


Figure 5.1: Execution plan for original query 5.1; generated by DB2 database

Figures 5.1 and 5.2 present the execution plans for original and optimized query 5.1. The original query was estimated to be performed within 8582.6 timeron units and needs one nested loops join with union, two hash joins and six table scans. The creation of the temporary table has been estimated for 8379.94 timeron units. On the other hand the query optimized using the method described in this chapter was estimated by the DBMS to be performed in 8377.54 timeron units and needs one nested loop join with union and four table scans. The creation of the temporary table in this case has been estimated for 8375.41 timeron units. Those plans show that the potential benefit of the optimization method for the query 5.1 lies within the reduction of the time needed to create the temporary CTE and the elimination of hash joins and two costly table scans.

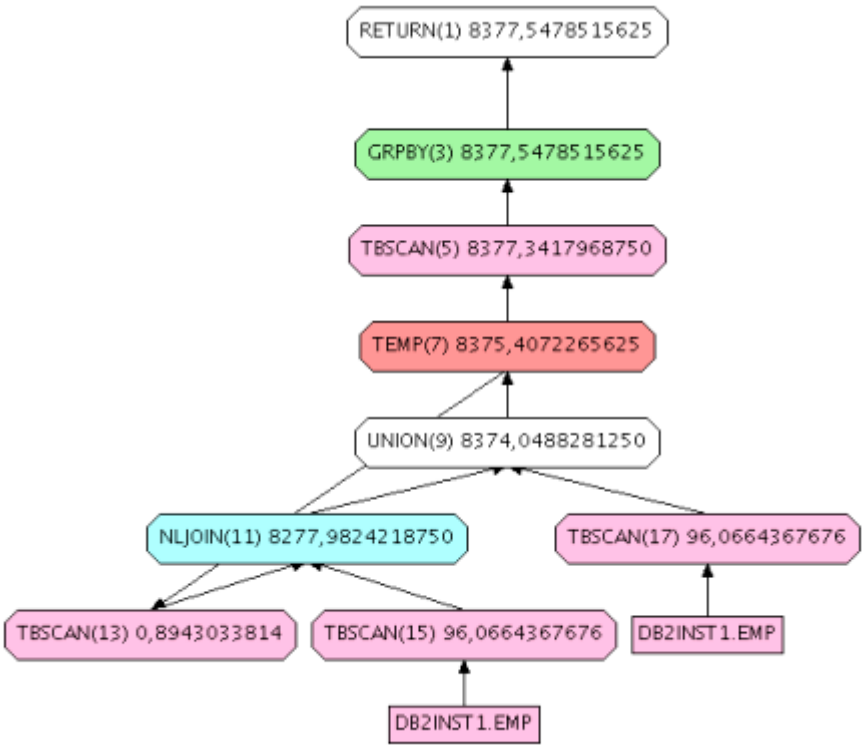


Figure 5.2: Execution plan for optimized query 5.1; generated by DB2 database

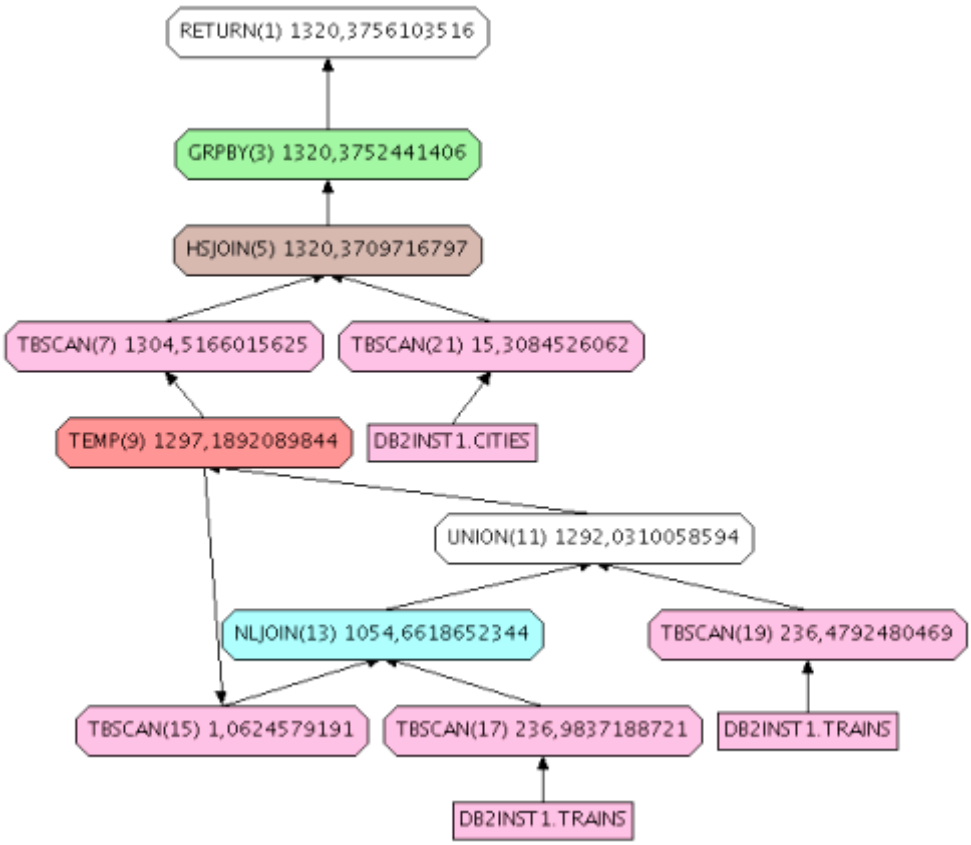


Figure 5.3: Execution plan for original query 5.2; generated by DB2 database

Now let us analyze execution plans for query 5.2 placed on figures 5.3 and 5.4. The original query was estimated to be performed within 1320.38 timeron units with the creation of temporary table estimated for 1297.19 timeron units. The optimized query was estimated for 1316.04 timeron units with its temporary table estimated for 1310.84 timeron units. This query in both cases needs one nested loops join with union, one hash join and five table scans. The benefit that comes out of the optimization strategy in this case is that the hash join operates on a smaller amount of data.

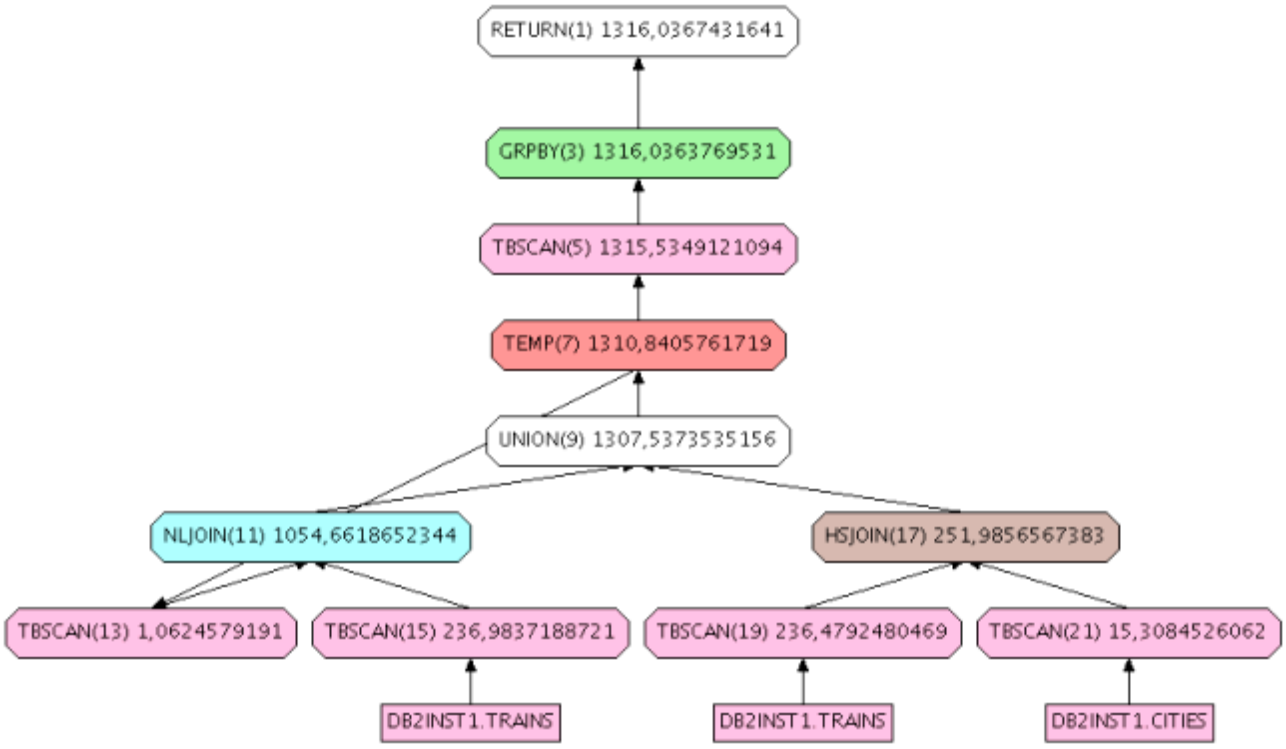


Figure 5.4: Execution plan for optimized query 5.2; generated by DB2 database

```
WITH  destinations (origin, departure, arrival, connections) AS
   (SELECT a.departure, a.departure, a.arrival, 1
           FROM trains a, cities c
           WHERE a.departure = c.cid
           AND c.city = 'Warsaw';
    UNION ALL
    SELECT r.origin, b.departure, b.arrival, r.connections + 1
           FROM destinations r, trains b
           WHERE r.arrival = b.departure
           AND r.connections < I )
SELECT count(*)
   FROM destinations e;
```

Query 5.3: Query 5.2 subjected to the pushing predicates technique

One more aspect of optimization should be considered. The optimization strategy presented in this chapter is similar to other, well known optimization technique called the magic sets [Ullm86, Ullm89]. Both are rewriting algorithms attempting to reduce the size of intermediate results. However while the magic set transformation operates only with equality and inequality comparisons, the pushing predicates technique allows for all kinds of predicates including inequality, greater-lesser comparisons and all other comparison operators. There are also other differences. In magic sets filtering is performed by creating additional tables and additional joins that keep only relevant tuples at each iteration. The pushing predicates technique on the other hand attempts to filter out irrelevant tuples at the first step of recursion. However, both optimization techniques may be used together.

5.2.5 Summary

The methods presented in this chapter deal with recursion problems. Suggested algorithms include reduction of intermediate structures for recursive SBQL queries and selecting the predicates which can be pushed into the SQL's recursive CTE. The condition that needs to be satisfied in the second case is the existence of tree invariant. The benefit of the usage of our method depends on the selectivity of the predicates being pushed and the recursion depth. A highly selective filter condition which may indirectly reduce the amount of recursion steps will improve the evaluation time in a significant way. Even experiments with small tables proved the high potential of the method, since for such small number of rows the reduction of the execution time is substantial.

The algorithm of predicate pushing for recursive SQL queries may be applied to both recursive Common Table Expressions and during inlining of recursive view definitions. It also is one of the key elements of optimization of object-relational mapping described in [Burz10a]

6 Final Conclusions

In this thesis we have shown four novel optimization techniques. Their main goal was to reduce the resource consumption occurring during query execution and/or reduce the time needed to calculate the result. All four techniques utilize query rewriting rules to achieve their goal. Using commercial DBMS's and PySBQL experimental platform we have shown that the presented optimization is profitable in both reduction of memory consumption and reduction of execution time. The SBQL language has been implemented in a number of systems including few European projects. Some systems and projects are still under development and there is a need for optimization algorithms. Presented algorithms have been developed bearing in mind those needs and algorithms that have already been implemented. The research on optimization of recursive SQL queries was part of a bigger study on the subject, which included comparison and efficiency testing of various implementations. This study revealed some of the weak points of evaluation of the recursive queries that could be fixed with proper evaluation and optimization algorithms. The technique developed for SQL language have shown it big potential during the experimental tests and inspired additional research on object-relational mappings of recursive SQL queries. Development of this algorithm together with the work on the deforestation algorithm inspired the work on other two presented optimization methods.

An interesting line for future research may be application of deforestation algorithms developed for stack based query languages family to the NoSQL databases equipped with map\reduce systems and a PySBQL wrapper. It also seems promising to combine research on deforestation and distributivity with parallelization of query evaluation. Last, but not least is research on benefits and applications of rewriting of recursive queries to their linear equivalents.

7 Bibliography

[Abit95] S. Abiteboul, R. Hull and V. Vianu. Foundations of Databases. Addison-Wesley 1995, ISBN 0-201-53771-0

[Abit97] S. Abiteboul, D. Quass, J. McHugh, J. Widom and J. Wiener. The Lorel Query Language for Semistructured Data. Int. J. on Digital Libraries 4(1/1), 1997, pages 68-88.

[Afan08] L. Afanasiev, T. Grust, M. Marx, J. Rittinger and J. Teubner. An Inflationary Fixed Point Operator in XQuery. In Proc. ICDE 2008. IEEE, 2008, pages 1504-1506

[Afan09] L. Afanasiev, T. Grust, M. Marx, J. Rittinger and J. Teubner. Recursion in XQuery: put your distributivity safety belt on. In Proc. EDBT 2009. ACM, 2009, pages 345-356

[Alme06] J. M. Almendros-Jiménez, A. Becerra-Terón and F. J. Enciso-Baños. Magic Sets for the XPath Language. Journal of Universal Computer Science 12(11), 2006, pages 1651—1678

[Apt86] K. R. Apt, H. A. Blair, A. Walker. Towards a Theory of Declarative Knowledge. Foundations of Deductive Databases and Logic Programming. Morgan Kaufmann 1988, ISBN 0-934613-40-0, pages 89-148

[Bamf09] R. Bamford, V. R. Borkar, M. Brantner, P. M. Fischer, D. Florescu, D. A. Graf, D. Kossmann, T. Kraska, D. Muresan, S. Nasoi and M. Zacharioudaki: XQuery Reloaded. In Proc. VLDB 2(2), Morgan Kaufmann Publishers, 2009, pages 1342-1353

[Beer87] C. Beeri and R. Ramakrishnan. On the Power of Magic. In Proc. ACM SIGACT-SIGMOD-SIGART PODS 1987. ACM, 1987, pages 269-284

[Bene08] M. Benedikt and C. Koch. XPath leashed. ACM Comp. Surveys, 41(1), 2008.

[Bonc06] P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger and J. Teubner: MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In Proc. SIGMOD '06. ACM, 2006, pages 479-490.

[Bran07] M. Brantner, N. May, G. Moerkotte. Unnesting Scalar SQL Queries in the Presence of Disjunction. In Proc. ICDE 2007, IEEE, Istanbul, 2007, pages 15-20

[Bres00] S. Bressan, C. H. Goh, N. Levina, S. E. Madnick, A. Shah and M. Siegel. Context Knowledge Representation and Reasoning in the Context Interchange System. Appl. Intell. (APIN) 13(2), Springer-Verlag, 2000, pages 165-180.

[Bris06] N. R. Brisaboa, A. Fariña, G. Navarro and J. R. Paramá. Chase of recursive queries. In Proc. Ershov Memorial Conference. LNCS 4378. Springer-Verlag, Berlin, 2007, pages 112-123.

[Burz07] M. Burzańska and P. Wiśniewski. PySBQL - Python-Like Query Language Constructed Using Stack Base Approach. Annales UMCS, Informatica, 2007, pages 143-151

[Burz09] M. Burzańska, K. Stencel and P. Wiśniewski. Pushing Predicates into Recursive SQL Common Table Expressions. In Proc. ADBIS 2009, LNCS 5739, Springer-Verlag, 2009, pages 194-205

[Burz09a] M. Burzańska and P. Wiśniewski. L-Value and R-Value Concept - Proposition to Solve Ref & Deref Chaos in SBQL Languages Family. Pol. J. Environ. Stud. Vol. 18 no. 3B, 2007, pages 143-151

[Burz10] M. Burzańska, K. Stencel and P. Wiśniewski. Intermediate Structure Reduction Algorithms for Stack Based Query Languages. In Proc. ASEA'10, CCIS 1(117), Springer-Verlag, 2010, pages 317-326

[Burz10a] M. Burzańska, K. Stencel, P. Suchomska, A. Szumowska, and P. Wiśniewski. Recursive Queries Using Object Relational Mapping. In Proc. FGIT'10, LNCS 6485, Springer-Verlag, 2010, pages 564-576

[Care00] M. Carey, J. Kiernan, J. Shanmugasundaram, E. Shekita, and S. Subramanian. SilkRoute: A Framework for Publishing Relational Data in XML. In Proc. VLDB. Morgan Kaufmann Publishers, 2000, pages 646–648.

[Catt96] R. G. G. Cattell. The Object Database Standard: ODMG-93 (Release 1.2). Morgan Kaufmann Publishers, 1996

[Ceri89] S. Ceri, G. Gottlob and L. Tanca. What You Always Wanted to Know About Datalog (And Never Dared to Ask). IEEE Trans. on Knowl. and Data Eng. 1(1), 1989, pages 146-166.

[Chau98] S. Chaudhuri. An overview of query optimization in relational systems. In Proc. PODS '98. ACM, New York, 1998, pages 34-43.

[Codd70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. Commun. ACM (CACM) 13(6). ACM, 1970, pages 377-387

[Codd72] E. F. Codd. Relational Completeness of Data Base Sublanguages. Database Systems. Prentice Hall and IBM Research Report RJ 987, San Jose, California, 1972,

[Date95] C. J. Date and H. Darwen. The third manifesto. In SIGMOD Rec. 24(1), ACM, 1995, pages 39-49.

[Daya87] U. Dayal. Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates and Quantifiers. In Proc. of VLDB. Morgan Kaufmann Publishers, 1987, pages 197-208

[Deck02] H. Decker. Translating advanced integrity checking technology to SQL. In Database integrity: challenges and solutions. Idea Group Publishing, 2002, pages 203–249

[Denn91] S. van Denneheuvel, K. L. Kwast, G. R. Renardel de Lavalette, E. Spaan. Query optimization using rewrite rules. In Proc. RTA '91. Springer-Verlag, New York, 1991, pages 252-263.

[Deut03] A. Deutsch and V. Tannen. Reformulation of XML Queries and Constraints. In Proc. ICDT, LNCS 2572, Springer-Verlag, 2003, pages 225–241

[Diet87] S. W. Dietrich: Extension Tables: Memo Relations in Logic Programming. In Proc. SLP '87, IEEE Computer Society, Washington, 1987, pages 264-272

[Elhe07] M. Elhemali, C. A. Galindo-Legaria, T. Grabs and M. M. Joshi. Execution strategies for SQL subqueries. In Proc. ACM SIGMOD '07. ACM, New York, 2007, pages 993-1004.

[Fega98] L. Fegaras. Query un-nesting in object-oriented databases. In Proc. ACM SIGMOD '98. ACM, New York, 1998, pages 49-60.

[Fega00] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. ACM Trans. Database Syst. 25(4). ACM, 2000, pages 457-516.

[Fern02] M. Fernandez, Y. Kadiyska, D. Suciu, A. Morishima, and W. C. Tan. XPERANTO: A Middleware for Publishing Object-Relational Data as XML Documents. In Proc. ACM TODS, 27. ACM, 2002, pages 438–493

[Flor04] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey and A. Sundararajan. The BEA Streaming XQuery processor. VLDB Journal 13(3). Morgan Kaufmann Publishers, 2004, pages 294–315.

[Fomi06] A. Fomichev, M. Grinev, and S. D. Kuznetsov. Sedna: A Native XML DBMS. In SOFSEM, 2006, pages 272-281

[Gall78] H. Gallaire and J. Minker (Eds.). Logic and Data Bases, Symposium on Logic and Data Bases, Plemum Press, New York, 1978, ISBN 0-306-40060-X

[Gans87] R. A. Ganski and H. K. T. Wong. Optimization of Nested SQL Queries Revisited. In Proc. ACM SIGMOD. ACM, San Francisco, 1987, pages 23-33.

[Ghaz06] A. Ghazal, A. Crolotte, D. Y. Seid: Recursive SQL Query Optimization with k-Iteration Lookahead. In Proc. DEXA 2006. LNCS 4080, Springer-Verlag, Kraków, 348-357

[Gill93] A. J. Gill, J. Launchbury and S. L. P. Jones. A short cut to deforestation. In Proc. FPCA, 1993, pages 223–232

[Gill96] A. J. Gill. Cheap deforestation for non-strict functional languages. PhD thesis, The University of Glasgow (1996)

[Godf94] P. Godfrey J. Minker and L. Novik. An Architecture for a Cooperative Database System. In Proc. ADB '94, LNCS 819, Springer Verlag, Vadstena, Sweden, 1994, pages 3-24

[Gott05] G. Gottlob, C. Koch, R. Pichler, and L. Segoufin. The complexity of Xpath query evaluation and XML typing. Journal of the ACM, 52(2), 2005, pages 284–335.

[Gotz09] M. Gotz, C. Koch, and W. Martens. Efficient algorithms for descendant-only tree pattern queries. Information Systems, 34(7), 2009, pages 602–623, 2009.

[Grin04] M. N. Grinev, D. Lizorkin. XQuery Function Inlining for Optimizing XQuery Queries. In Proc. ADBIS '04, 2004

[Grin05] M. N. Grinev and P. Pleshachkov. Rewriting-Based Optimization for XQuery Transformational Queries. In Proc. IDEAS '05. IEEE Computer Society, Washington, 2005, pages 163-174.

[Grus98] T. Grust, M. H. Scholl. Query deforestation. Technical report, Database Research Group, University of Konstanz, 1998

[Grus04] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In Proc. VLDB. Morgan Kaufmann Publishers, 2004, pages 252–263.

[Gupt92] A. Gupta and I. S. Mumick. Magic-sets transformation in nonrecursive systems. In Proc. PODS '92. ACM, New York, 1992, pages 354-367

[Haji05] E. Hajiyev, M. Verbaere, 0. de Moor and K. de Volder. CodeQuest: querying source code with datalog. In Proc. OOPSLA '05. ACM, New York, 2005, pages 102-103.

[Halv04] A. Halverson, V. Josifovski, G. M. Lohman, H. Pirahesh and M. Mörschel. ROX: Relational over XML. In Proc. VLDB. Morgan Kaufmann Publishers, Toronto, 2004, pages 264-275

[Hell92] J. M. Hellerstein. Predicate Migration: Optimizing Queries with. Technical Report. UMI Order Number: S2K-92-13., University of California at Berkeley, 1992

[Jeze88] K. Jezek and V. Toncar. Experimental deductive database. In Workshop on Information Systems Modelling, 1988, pages 83-90

[Jigy06] S. Jigyasu, S. Banerjee, V. Borkar, M. Carey, K. Dixit, A. Malkani and A. Thatte. SQL to XQuery Translation in the AquaLogic Data Services Platform. In Proc ICDE '06. IEEE Computer Society, Washington, 2006, page 97.

[Joha01] P. Johann. Short cut fusion: Proved and improved. In LNCS 2196, Springer-Verlag, 2001, pages 47–71

[Jone01] S. P. Jones, A. Tolmach and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In Haskell Workshop, ACM SIGPLAN, 2001, pages 203–233

[Kemp94] A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimizing disjunctive queries with expensive predicates. In Proc. ACM SIGMOD 23(2), ACM, 1994, pages 336–347.

[Kim82] W. Kim. On Optimizing an SQL-like Nested Query. ACM TODS, Vol 9 (3), 1982.

[Koym90] K. Koymen. A Datalog interface for SQL (abstract). In Proc. ACM CSC '90. ACM, New York, 1990, page 422.

[Kris04] R. Krishnamurthy, P. Kaushik, and J. Naughton. Efficient XML-to-SQL Query Translation: Where to Add the Intelligence? In Proc. VLDB. Morgan Kaufmann Publishers, 2004, pages 144–155

[Levy94] A.Y. Levy, I. S. Mumick and Y. Sagiv. Query Optimization by Predicate Move-Around. In Proc. VLDB. Morgan Kaufmann Publishers, San Francisco, 1994, pages 96-107.

[Kay06] M. Kay. Optimization in XSLT and XQuery. In Proc XML Prague, 2006

[Lent06] M. Lentner, K. Stencel and K. Subieta. Semi-strong Static Type Checking of Object-Oriented Query Languages, In Proc. SOFSEM '06, LNCS 3831, Springer-Verlag, 2006, pages 399 – 408

[Liu08] Z. H. Liu, A. Novoselsky, and V. Arora. Towards a Unified Declarative and Imperative XQuery Processor. IEEE Data Engineering Bulletin, 31, 2008.

[Maie88] D. Maier and D. S. Warren. Computing with Logic: Logic Programming with Prolog. Benjamin-Cummings, 1988.

[Mano01] I. Manolescu, D. Florescu and D. Kossmann. Answering XML Queries on Heterogeneous Data Sources. In Proc. VLDB. Morgan Kaufmann Publishers, 2001.

[May06] N. May, S. Helmer and G. Moerkotte. Strategies for query un-nesting in XML databases. ACM Trans. Database Syst. 31(3), 2006, pages 968-1013.

[Mumi94] I. S. Mumick and H. Pirahesh. Implementation of magic-sets in a relational database system. In Proc. ACM SIGMOD 23(2), ACM, 1994, pages 103-114.

[Mura92] M. Muralikrishna. Improved un-nesting Algorithms for Join Aggregate SQL Queries. In Proc. VLDB. Morgan Kaufmann Publishers, Vancouver, 1992.

[Nejd87] W. Nejdl. Recursive Strategies for Answering Recursive Queries - The RQA/FQI Strategy. In Proc. VLDB. Morgan Kaufmann Publishers, San Francisco, 1987, pages 43-50.

[Norm03] N. May, S. Helmer and G. Moerkotte. Three Cases for Query Decorrelation in XQuery. LNCS 2824, Springer-Verlag, 2003, pages 70-84

[Ohor07] A. Ohori and I. Sasano: Lightweight fusion by fixed point promotion. In Proc. ACM SIGPLAN-SIGACT POPL 2007. ACM, 2007, pages 143-154

[OMG07] OMG Object Database Technology Working Group: Next-Generation Object Database Standardization, OMG White paper, 2007. Available at `http://www.omg.org/docs/mars/07-09-13.pdf`

[Ordo05] C. Ordonez. Optimizing recursive queries in SQL. In Proc. SIGMOD '05. ACM, 2005, pages 834-839

[Ordo10] C. Ordonez. Optimization of Linear Recursive Queries in SQL. IEEE Trans. on Knowl. and Data Eng., IEEE, 2010, pages 264-277

[Ozca08] F. Özcan, N. Seemann, and L. Wang. XQuery Rewrite Optimization in IBM DB2 pureXML. IEEE Data Engineering Bulletin, 31(4), 2008, pages 25-32

[Papa95] Y. Papakonstantinou, H. Garcia-Molina J. and Widom J. Object exchange across heterogeneous information sources. In Proc. 11th Int. Conf. on Data Engineering, 1995, pp. 251–260.

[Pary09] P. Parys. XPath evaluation in linear time with polynomial combined complexity. In Proc. PODS '09. ACM, New York, 2009, pages 55-64

[Piec06] T. Pieciukiewicz, K. Stencel and K. Subieta. Object-Oriented Programming with Recursive Queries. Databases and Applications 2006, pages 228-233

[Piec08] T. Pieciukiewicz, K. Stencel and K. Subieta. Recursive Query Processing in SBQL. In Proc. ICOODB, 2008, pages 57-76

[Piec10] T. Pieciukiewicz. Recursive Queries in Databases. PhD thesis, Polish-Japanese Institute of Information, Warsaw, 2010

[Plod00] J. Płodzień. Optimization Methods in Object Query Languages. PhD thesis, Institute of Computer Science, Polish Academy of Science, Warsaw, 2000,

[PLY] PLY (Python-Lex-Yacc) library. Available at `http://www.dabeaz.com/ply/`

[Przy10] P. Przymus, A. Boniewicz, M. Burzanska and K. Stencel. Recursive query facilities in relational databases: a survey. In proc. DTA/BSBT'10, CCIS 1(118), Springer-Verlag, 2010, pages 89-99

[Przy86] T. Przymusinski. On the semantics of stratified deductive databases. In Proc. Workshop Foundations Deductive Databases Logic Programming, Washington, 1986, pages 433-443.

[Python] Python programming language. Available at `http://www.python.org/`

[Rama88] R. Ramakrishnan, "Magic templates, A spellbinding approach to logic evaluation," in Proc. Logic Programming Conf, 1988

[Rama91] R. Ramakrishnan and S. Sudarshan. Top-Down vs. Bottom-Up Revisited. In Proc. ISLP'91, 1991, pages 321-336

[Ross91] K. A. Ross. Modular acyclicity and tail recursion in logic programs. In Proc. PODS'91, ACM, 1991, pages 92-101

[Ross96] K. A. Ross. Tail recursion elimination in deductive databases. ACM Trans. Database Syst. 21(2), 1996, pages 208-237.

[Rubi10] M. Rubio-Sánchez, M. Tail recursive programming by applying generalization. In Proc. ITiCSE '10. ACM, New York, 2010, pages 98-102.

[Saxon] M. Kay. Saxon: The XSLT and XQuery processor. Available at `http://saxon.sourceforge.net/`

[Schn08] L. Schneider and D. Burleson. Advanced Oracle SQL Programming: The Expert Guide to Writing Complex Queries. Rampant TechPress, 2008.

[Seki89] H. Seki. On the power of Alexander templates. In Proc. PODS'89, 1989, pages 150–159

[Sesh96] P. Seshadri, H. Pirahesh and T. Y. C. Leung. Complex Query Decorrelation. In Proc. of ICDE, 1996.

[Shoe93] K. Shoens, A. Luniewski, P. Schwarz, J. Stamos and J. Thomas. The Rufus system: Information organization for semi-structured data. In Proc. VLDB. Morgan Kaufmann Publishers, 1993, pages 97–107.

[Sten06] K. Stencel. Półmocna kontrola typów w językach programowania baz danych. Editors of the PJWSTK, Warsaw, 2006 (in Polish)

[Ston75] M. Stonebraker. Implementation of integrity constraints and views by query modification. In Proc. SIGMOD '75. ACM, New York, 1975, pages 65-78.

[Subi90a] K. Subieta. LOQIS: The Object-Oriented Database Programming System. In Proc. East/West Database Workshop 1990, Springer, Kiev, USSR, 1990, pages 403-421.

[Subi90b] K. Subieta, M. Missala, K. Anacki. The LOQIS System. Technical Report 695, Institute of Computer Science Polish Academy of Sciences, Warsaw, Poland, 1990.

[Subi94] K. Subieta, C. Beeri, F. Matthes and J. Schmidt. A Stack-Based Approach to Query Languages. In Proc. East/West Database Workshop, 1994, pages 159-180.

[Subi96] K. Subieta. Object-Oriented Standards: Can ODMG OQL be Extented to a Programming Language? In Proc. CODAS, World Scientific, Japan, 1996, pages 459-468

[Subi04] K. Subieta. Theory and Construction of Object-Oriented Query Languages. Editors of the PJWSTK, Warsaw, 2004 (in Polish)

[SBQL] Stack Based Query Language: Recursive Operators. Available at http://www.sbql.pl/Topics/SBQL%20Recursive.html

[Ullm86] J. D. Ullman, F. Bancilhon, D. Maier and Sagiv Y. Magic sets and other strange ways to implement logic programs. In Proc. SIGACT-SIGMOD, ACM, 1986, pages 1–15.

[Ullm88] J. D. Ullman. Principles of Database and Knowledge-Base Systems, Volume I. Computer Science Press 1988, ISBN 0-7167-8158-1

[Ullm89] J. D. Ullman. 1989. Bottom-up beats top-down for datalog. In Proc. PODS '89. ACM, New York, NY, 1989, pages140-149

[Ullm95] J. D. Ullman and R. Ramakrishnan. A survey of research in deductive database systems. J. Logic Program. 23(2), 1995, pages 125--150.

[Vidh10] P. M. Vidhya and P. Samuel. Query translation from SQL to XPath. In Proc. NaBIC 2009, IEEE, 2010, pages 1749—1752,

[Viei87] L. Vieille. A Database-Complete Proof Procedure Based on SLD-Resolution. In Proc. ICLP 1987, pages 74-103

[Voig08] J. Voigtländer. Semantics and Pragmatics of New Shortcut Fusion Rules. In Proc. FLOPS 2008, LNCS 4989, Springer-Verlag, 2008, pages 163-179,

[Wadl90] P. Wadler. Deforestation: Transforming programs to eliminate trees. Theor. Comput. Sci. 73(2), 1990, pages 231–248

[Wisl07] J. Wislicki. An object-oriented wrapper to relational databases with query optimisation. PhD Thesis, Polish-Japanese Institute of Information, Warsaw, 2007

[XML1.0] Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C Recommendation 26 November 2008. Available at http://www.w3.org/TR/xml/

[XML1.1] Extensible Markup Language (XML) 1.1 (Second Edition). W3C recommendation 16 August 2006. Available at http://www.w3.org/TR/xml11/

[XMLQL] XML-QL: A Query Language for XML. Submission to the W3C 19 August 1998 Available at http://www.w3.org/TR/NOTE-xml-ql/

[XPath1] XML path language (XPath), version 1.0. W3C recommendation 16 November 1999. Available at http://www.w3.org/TR/xpath/

[XPath2] XML path language (XPath), version 2.0, W3C recommendation 23 January 2007. Available at http://www.w3.org/TR/xpath20/

[XQuery] XQuery. XQuery 1.0: An XML query language. Available at

http://www.w3.org/TR/Xquery

[XSLT] XSL Transformations (XSLT) Version 2.0. W3C Recommendation 23 January 2007. Available at http://www.w3.org/TR/xslt20/

[Yan94] W. P. Yan and P. Larson. Performing Group-By before Join. In Proceedings of the Tenth international Conference on Data Engineering. IEEE Computer Society, Washington, DC, 1994, pages 89-100.