

Nicolaus Copernicus University
Faculty of Mathematics and Computer Science

Marta Jadwiga Rogińska
Album nr: 158 141

Master Thesis
Computer Science

PySBQL

Construction and Implementation of the Object Query Language

Work written under the guidance of
Dr. Piotr Wiśniewski
Faculty of Mathematics and Computer Science

Toruń 2006

1. Introduction.....	5
1.1. Basic Concepts.....	7
2. Basics Of Natural And Computer Linguistics.....	10
2.1. Syntax.....	11
2.2. Semantics.....	12
2.3. Pragmatics.....	13
2.4. Lexemes and Linguistic Units.....	14
3. Data Models.....	16
3.1. Hierarchical Data Model.....	18
3.1.1. Basic Concepts.....	18
3.1.2. Virtual Parent-Child Relationship.....	19
3.1.3. DDL and DML in the Hierarchical Data Model.....	20
3.1.4. Summary.....	24
3.2. Network Data Model.....	25
3.2.1. Basic Concepts.....	25
3.2.2. Constraints on sets.....	27
3.2.3. DDL and DML in the Network Data Model.....	29
3.2.4. Summary.....	31
3.3. Relational Data Model.....	32
3.3.1. Basic Concepts.....	32
3.3.2. Normal Form.....	33
3.3.3. Operations on Relations and the SQL.....	34
3.3.3. Codd's 12 rules.....	38
3.3.4. Summary.....	41
3.4. Object-relational Data Model.....	42
Nested relations.....	42
Triggers.....	43
Stored procedures.....	43
Tables as objects.....	43
3.5. ODMG's Object model.....	45
3.5.1. ODMG manifest.....	45
3.5.2. Object Model.....	46
3.5.3. Object Query Language.....	47
3.5.4. Summary.....	48
4. Approaches to Query Languages.....	49
4.1. Environment Stack.....	51
4.2. Name Binding.....	51
4.3. Nesting.....	52
4.4. Results Returned by SBQL Queries.....	53
4.5. Query Result Stack.....	54
4.6. M0 Model.....	55
4.7. SBQL for the M0 Model.....	56

4.8. Comparison of SBQL, SQL and OQL Queries.....	58
5. PySBQL.....	59
5.1. Main Concepts Behind the Language.....	59
5.2. Why Python.....	59
5.3. PySBQL grammar.....	60
5.4. Examples of PySBQL Queries and Statements.....	63
5.5. Construction of the Parse Tree.....	65
5.6. Source Code Evaluations.....	65
5.7. Results of Expressions and the Result Stack.....	66
5.8. Environmental Stack.....	66
5.9. Differences between SBQL and PySBQL.....	66
5.10. Differences between PySBQL and Python.....	67
5.11. Additional Modules.....	68
6. Content of the CD.....	69
7. Bibliography.....	70

1. Introduction

In the last years the tendency towards objectivity is being observed. The relational data model does not fulfill the needs of the databases' users. Also the alternative for the SQL query language was sought. What's more, ever since the first databases and first query languages were created there was a need to combine them with a programming language. Many attempts to embed a query language inside a programming language like C or Java only proved that is not a solution, and the proper way is to construct a language from the start combining the attributes of a query language and a programming language

The purpose of this work was to implement a fully operational query and programming language, that could be used with objectable databases, mainly YODA database management system created by Michał Burzański This language would be based on the stack approach and would be a functional tool for writing applications as well as communicating with database. It should also be easy to learn and its usage modeled on the usage of natural languages.

This paper is divided into 6 chapter. The first chapter is this introduction. The second presents the brief concepts of linguistics, the definition of the terms "meaning" and "lexical unit". It introduces the basis on which any computer language should be built. The third chapter presents concepts of the five most popular data models with the corresponding query languages. It is intended to be a brief history of the development of databases and the query languages. The fourth chapter presents the concepts of the stack bases approach (SBA) and stack based query language (SBQL) developed by prof. Kazimierz Subieta. Those concepts were the main building blocks for the construction of PySBQL, which is described in the fifth chapter. In this chapter the main differences between PySBQL and SBQL are pointed out. It also presents the possibilities of future further development of PySBQL modules. The source code of the language, including the source code of the YODA DBMS, is placed on the CD disc added to this paper. The content of the CD if described in the chapter 6.

The PySBQL has been implemented in the Java language using tools JavaCC (Java Compiler Creator) – parser/scanner creation tool and JJTree – preprocessor for JavaCC that inserts parse tree building actions in the configuration file.

Java language has been chosen because it grants the identical code operation on any of the system platforms that have the JVM. Also the source code written in Java is readable and easy to maintain.

JavaCC is a very popular tool for generation of LL(k) parsers. It generates files that are easy to modify and to use. The big advantage of this tool is that the specification of the scanner and parser is placed in one file of the readable form. The grammar specification may be written in the BNF notation (see sub-chapter 1.1 for more information). Its preprocessor – JJTree – tool for the generation of abstract syntax trees (AST) allowed for the concentration on the grammar details instead of concentrating on the implementation details. Those tools were chosen after analyzing the structure of compilers and interpreters, and after comparing available tools.

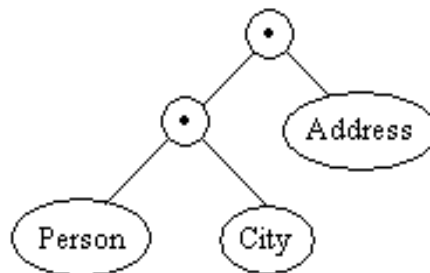
To present the JJTree let us consider following BNF (Backus–Naur form described in chapter 2.1) rule:

$$\text{DotE} \rightarrow \text{DotE} . \text{SimpleExpression}$$

The corresponding code written in JJTree (taken from Sbql.jjt file) is as follows:

```
void DotE() #void: { }  
{  
    SimpleE() ( <DOT> SimpleE() #DotE(2) ) *  
}
```

#DotE(2) means that the node DotE may have only two children, so if the parser would encounter a string “Person.Address.City” the following tree would be created:



The action that should be performed during the evaluation of a node DotE is implemented in the file Evaluator.java in the function wykonaj(). In order to find it, a case statement labeled SbqlTreeConstants.JJTDOTE must be found.

1.1. Basic Concepts

This sub-chapter defines the main terms used in this paper.

1.1.1. Stacks, Queues and Maps

Queues and stacks are dynamic data structures. They both always contain the ordered set of elements.

The stack is a linear data structure, in which the element is being added behind the recently added element, to the top of stack. The only accessible element is the one last added and so it is the first to be removed from the stack (LIFO – Last In, First Out)

In the queue an element is being added to the end of the ordered set of the queue's elements, and the only available element is the first element, and only it may be removed from the queue. (FIFO –First in First Out)

A dictionary (map) is an abstract data type composed of a collection of keys and a collection of values, where each key is associated with one value, and the key cannot be duplicated (values may)

1.1.2. Graphs

A **graph** or **undirected graph** G is an ordered pair $G := (V, E)$ that is subject to the following conditions:

- V is a set of vertices or nodes,
- E is a set of unordered pairs of distinct vertices, called edges or lines.
- The vertices belonging to an edge are called the ends, endpoints, or end vertices of the edge.

V (and hence E) are usually taken to be finite sets, and many of the well-known results are not true (or are rather different) for infinite graphs because many of the arguments fail in the infinite case. The example graph with $V=\{1,2,3,4,5\}$ and $E=\{(m,n) ; m+n=4 \vee m+n=7\}$ is shown on the figure 1.1.

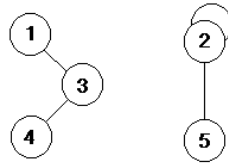


Figure 1.1. A sample graph

A **directed graph** or **digraph** G is an ordered pair $G := (V, A)$ with

- V , a set of vertices or nodes,
- A , a set of ordered pairs of vertices, called directed edges, arcs, or arrows. An edge $e = (x, y)$ is considered to be directed from x to y ; y is called the head and x is called the tail of the edge.

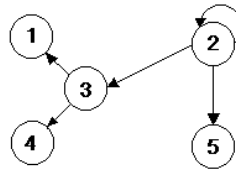


Figure 1.2 A sample digraph

1.1.3. Tree

A **tree** is an undirected simple graph G that satisfies any of the following equivalent conditions:

- G is connected and has no simple cycles.
- G has no simple cycles and, if any edge is added to G , then a simple cycle is formed.
- G is connected and, if any edge is removed from G , then it is not connected anymore.
- G is connected and the 3-vertex complete graph K_3 is not a minor of G .
- Any two vertices in G can be connected by a unique simple path.

If G has finitely many vertices, say n of them, then the above statements are also equivalent to any of the following conditions:

- G is connected and has $n - 1$ edges.
- G has no simple cycles and has $n - 1$ edges.

An *undirected* simple graph G is called a **forest** if it has no simple cycles.

A *directed* tree is a directed graph which would be a tree if the directions on the edges were ignored. Some authors restrict the phrase to the case where the edges are all directed towards a particular vertex, or all directed away from a particular vertex.

A tree is called a **rooted tree** if one vertex has been designated the root, in which case the edges have a natural orientation, towards or away from the root. Rooted trees, often with additional structure such as ordering of the neighbors at each vertex, are a key data structure in computer science

A *labeled* tree is a tree in which each vertex is given a unique label.

A *binary* tree is a tree in which each node has at most two sub-nodes.

2. Basics Of Natural And Computer Linguistics

During the process of analyzing the structure of communicates, with no regard towards whether they are sent to a computer or to a person, many new terms and definitions emerge. Such terms are e.g. *alphabet*, *letter*, *word*, *symbol*. A precise definition of those terms is needed.

Symbol is a non-definable term, just like a point or a line. **Alphabet** (*dictionary*) is a non-empty finite set consisting of symbols. Alphabet elements are usually called **letters** or **signs**.

Having the letters a_1, a_2, \dots, a_n from alphabet A , A^+ is the set of all finite strings $a_1 \dots a_n$, $n \geq 1$, created by concatenation of elements from A . A^* is a set A^+ with added an **empty string** (not containing any symbols) denoted by ϵ . It is a neutral element of an operation of concatenation: $\epsilon w = w \epsilon = w$, $w \in A^*$. The number n of the symbols in the string $w = a_1 \dots a_n$ is called a **length** of w and denoted by $|w|$. Thus $|\epsilon| = 0$. String w is called a *word* over the alphabet A .

In this paper we will often refer to the term “language”, but for what does it stand (besides the biological meaning). Citing from the Encyclopedia Britannica, term “language” stands for: “the words, their pronunciation, and the methods of combining them used and understood by a community”. In other words it is a general set of words with rules on how to combine them into proper sentences.

Computer science has also its definition of the term “language”: having a finite set X , every subset of a set X^* is called a (*formal*) **language** over the dictionary X . Elements of a language are called **words** or **sentences**. Both of those definitions may look different at a first sight, but they describe the same thing.

The general science of signs, their creation and meaning, is called **semiotics**. The most common division of semiotics into *semantics*, *syntax* and *pragmatics* was introduced in 1938 by Charles Morris in the work “*Foundations of the Theory of Signs*”. The criteria of this division are based on types of relations involving signs. And so *syntax* studies the relations between a sign and a sign inside a specific language. *Semantics* studies the relations between signs and reality, to which those signs refer. *Pragmatics* – studies the relations between the signs and their users (receiver, sender).

While explaining any language, in particular a language to be implemented on a computer, it is important to clearly separate those three aspects.

2.1. Syntax

Restricting only to the natural languages, or to computer languages, a specification of the objects of those disciplines is possible. For the natural languages the place of **syntax**, as a main domain of linguistics, is taken by *grammar*. It studies the system of rules determining the ways of inflection and creation of vocabulary units, ways of joining them into constructions (groups, sentences).

Being limited only to computer languages it is spoken of a *syntax* of the language but in a narrower way. There also exist a term "*formal grammar*". In computer science a formal grammar is an abstract structure that describes a formal language precisely; it is a set of rules that mathematically delineates a (usually infinite) set of finite-length strings over a (usually finite) alphabet. Both computer language syntax and natural language grammar do not fully analyze the dependence between following sentences.

One more definition needs to be placed here:

In linguistics and computer science, a **context-free grammar (CFG)** is a formal grammar in which every production rule is of the form $V \rightarrow w$ where V is a non-terminal symbol and w is a string consisting of terminals and/or non-terminals. The term "context-free" comes from the fact that the non-terminal V can always be replaced by w , regardless of the context in which it occurs. A formal language is context-free if there is a context-free grammar that generates it.

Context-free grammars are powerful enough to describe the syntax of most programming languages; in fact, the syntax of most programming languages are specified using context-free grammars. On the other hand, context-free grammars are simple enough to allow the construction of efficient parsing algorithms which, for a given string, determine whether and how it can be generated from the grammar. BNF (Backus-Naur Form) is the most common notation used to express context-free grammars.

Examples of the notations are placed below:

Traditional notation:

<sentence> → < noun phrase > < verb phrase >
<noun phrase> → <adjective> < noun phrase >
< noun phrase > → <noun>
<noun> → boy

Backus–Naur form (BNF):

<expression> → <expression> + <expression>
<expression> → <expression> * <expression>
<expression> → (<expression>)
<expression> → **id**

id represents the arguments of the operators “+” and “*”.

The Extended Backus–Naur Form (EBNF) is a BNF expanded by the usage of regular expressions

2.2. Semantics

Semantics studies the system of rules that assign to specific linguistic expressions (of a varying complexity level) the representation of their contents. To fully understand the issue of semantics, defining of a certain domain of meanings is needed. It may be defined in a variety of ways, depending on a target and on a receiver of the definition. For people working with computer languages semantics may describe a certain way of expressing a program (e.g. written in the C language) in a predefined meta-language.

While describing semantics it is necessary to introduce the term *meaning*. It is a term widely discussed. Basically there are two opposite approaches: one of them identifies the meaning of a linguistic expression with its content, whereas the other combines the word “meaning” into at least two word terms like *descriptive meaning*, *grammatical meaning*, *cognitive meaning* and so on...

In this paper it will be assumed that the meaning of an expression E in a language L describes what is being passed on to the receiver using E; what the receiver found out on the basis that the sender used the expression E.

For example, the meaning of the expression “Johnny is eating falafel” is perceived as an act of consumption of a dish called “falafel”¹ performed by a person named Johnny, probably a young boy. Semantics of a certain application written in Java may be expressed by a sequence of instructions of an abstract machine (with an assumption that the semantics of this machine’s instructions is given and is a form of non-defined axiom)

In case of computer languages another matter must also be taken into consideration – the above example is understandable for a programmer (programmer, as a human, “understands” certain strings in an intuitive, informal way), but a computer in order to “understand” an expression E, must translate it into a sequence of orders, which it understands (the semantics of this sequence is given). It turns out that computer languages should be addressed both to a programmer and to computer. Also the mapping of language strings into certain actions of a computer should refer to the abstract terms and operations, rather than to a machine code. Inconsistency between informal understanding of language strings and a formal representation of this string into the actions of a computer is called a *semantic reef*. Semantics may be divided into *lexical semantics* and *semantics of a sentence*.

2.3. Pragmatics

The last of the semiotics discipline – **pragmatics** – does not undergo a full formalization. Pragmatics of a language determines its usage functions in the interpersonal interaction or in the interaction between a human and a machine. It studies the system of rules assigning linguistic expressions to specific classes of situations. Usage of the pragmatics’ rules depends on the will of the sender, who, accordingly to a situation, chooses those linguistic means, whose usage is determined by the linguistic convention applying to the situation of a given type. Pragmatics is explained by examples, various cases and analogy. The majority of language courses concerns its pragmatics, which is being explained on dozens of examples, counterexamples, explanations of the good and bad usage style.

In literature discussing the artificial languages pragmatics is often mixed with semantics which is being explained by the examples of usage accustomed to a fitted to a certain situation or the task. However pragmatics is not a good method of explaining semantics, because is not capable of faithfully describe the mutual connection of individual structures of the language, especially when dealing with its recursive definition. That it is why in this paper the goal will

¹ It is a kind of vegetarian hamburger

be to precisely describe languages using mostly semantic with only a limited help from pragmatics.

2.4. Lexemes and Linguistic Units

One of the basic concepts of a lexical semantics are terms lexeme and linguistic unit. Lexeme is defined as the indicator of the set of all forms that have the same referential value or the same regular referential diversity in the whole grammar class, meaning they have different grammar forms but the same semantic features. The set of grammatical forms “handm hands” is represented by a lexeme hand, and the set of grammatical forms “choose, chose, chosen” is represented by the lexeme “choose”. Lexical unit is a standard form of lexeme, that generalizes the features represented by the lexeme of the set and is undividable by meaning. This means that every lexical unit A must fulfill 3 conditions:

1. Separation between any two sequences X and Y in the context XAY – the meaning of he sequence A in the context X_1AY_1 is the same as in the context X_nAY_n . If, while studying contexts of this type, a context in which A has a different meaning is encountered, it indicates that either a new (of the same form) unit A' was found, or that A is part of a bigger unit, or a sequence A is a concatenation of a couple of lexical units.
2. Every expression that can take the place of X and Y must be an element of some unclosed class²
3. None of the elements of A fulfills the conditions laid on A.

For example:

Context 1:	This is placed	<i>dead</i>	ahead
	X_1	A	Y_1
Context 2:	Adam has a	<i>dead</i>	clock
	X_2	A	Y_2
Context 3:	I found a	<i>dead</i>	phone
	X_3	A	Y_3

² Unclosed class is a setoff elements, to which a general characteristics may be assigned. A closed class may be defined only by listing its elements

Sequence A from the context 1 presents a different lexical unit than in contexts 2 and 3.

Natural languages have the tendency to keep lexical units short, although they may be multi-worded. But then the words are short. This may be one of the causes why the SQL language (described in chapter 3.3) did not accomplish its goal of being close to natural language – its lexical units are often multi-worded (like `SELECT FROM`, `DELETE FROM`) and long. The next mistake, that is visible in the literature concerning computer languages, is that pragmatics is often confused with semantics, and the key words are confused with lexical units.

3. Data Models

Each database management system (DBMS) is based upon some model of data. The term “data model” exist mainly in two meanings: as a data architecture and as an integrated set of rules and constraints in the data.

As an architecture, data model is a set of general rules on using the data. An example of an architectonic data model is a relational model or an object model. A set of rules, that determine the data model, may be divided into three main parts:

1. Data definition – set of rules defining the structure of the data
2. Data manipulation – set of rules defining what operations may be done on the data
3. Data integrity – set of rules determining which states of the database are correct.

As a project, data model is seen as an integrated, independent from any implementation, set of rules and constraints on a specific data for a specific application. An example may be a model of data for an accountancy department, or a data model for a students record files.

This paper will briefly describe five architectonical data models:

1. Hierarchical model
2. Network model (also called the Graph model)
3. Relational model
4. Object-relational model
5. ODMG's³ Object model

Each DBMS also requires a way of accessing and manipulating data. First DBMSs used sets of very simple commands. They were called *record-at-a-time languages*, and usually consisted of only a few key words – commands. In the 1970's a group at IBM's San Jose research center developed a database system "System R" based upon, but not strictly faithful to, E. F. Codd's relational model. Structured English Query Language ("SEQUEL") was designed to manipulate and retrieve data stored in System R. The acronym SEQUEL was later condensed to SQL (pronounced *sequel* or *ess-cue-el*) because the word 'SEQUEL' was held as a trademark by the Hawker-Siddeley aircraft company of the UK. Although SQL was influenced by Codd's work, Donald D. Chamberlin and Raymond F. Boyce at IBM were the authors of the SEQUEL language design. Their goal was to create a language as similar in

³ Object Database Management Group

usage to natural language, as possible. And the first, small implementation of SQL, was indeed very similar. Based on the data model, SQL user could have built short “sentences” to “communicate” to DBMS. But as the research on SQL progressed it became more robust, and the additional key words, that make very simple SQL commands remind the user of an English language, when it comes to more complex demands contribute to the creation of big, hard to understand constructions, that are very far from natural language forms (which has a tendency to express thoughts and ideas in a short, compound form). That is because in the SQL lexemes are very often created out of two or even more words.

Other query languages that were developed parallel to SQL are QUEL (which was implemented in Ingres – the first “relational” DBMS) and Query-by-example – the first graphical query language. They will not be discussed in this paper.

When the object model became a subject of research, Object Data Management Group (ODMG) created a base concept of an Object Query Language (OQL). It was modeled after SQL and inherited big complexity of more advanced queries.

Nowadays there are a few projects on development of query languages that could be used with object databases. One of those languages is SBQL developed in Warsaw (Poland) by a group of scientist led by prof. Kazimierz Subieta. The main concept of this language will be presented in chapter 4 of this work.

This chapter presents the main concepts of the four general data models and an object-relational model. Each general data model is presented with a corresponding query language. Each of those query languages has its commands divided into 3 groups: Data Definition Language (DDL), Data Manipulation Language (DML) and Data Control Language (DCL).

- DDL is set of commands used for defining database schemes, and structures of data (records, tables).
- DML consists of commands used for retrieval, and modification of data.
- DCL commands are used to set constrains on the data.

3.1. Hierarchical Data Model

Hierarchical model of data was developed in the 1960's to model many types of hierarchical organizations that exist in the real world. First hierarchical database management system was IBM's Information Management System (IMS), developed for the Apollo Landing Project. There has been no official document presenting the standard for the hierarchical model. Instead several information management systems have been developed using hierarchical storage structures. This paper presents common principles behind the hierarchical model, but in some cases it will relate to the IMS, which was the dominant hierarchical system.

3.1.1. Basic Concepts

The hierarchical data model consists of two basic structures: a record and a parent-child relationship (PCR). *Records* are the means of storing data. Each one is a collection of *fields*, whereas each field stores only one data value (e.g. integer, real, string). Records of the same structure are gathered in *record types*. Record type has a name and a defined structure of named fields.

EMPLOYEE		
NAME	ADDRESS	SALARY

<u>Data item name</u>	<u>format</u>
NAME	CHARACTER 30
ADDRESS	CHARACTER 35
SALARY	REAL

Figure 3.1.1. A record type EMPLOYEE

A *parent-child relationship type* (PCR type) is a 1:N relationship between two record types. One of them is chosen as a *parent* record type, whereas the other is a *child* record type of the PCR type. An occurrence (instance) of PCR type consists of exactly one record of parent record type and a number of records (zero or more) of the child record type. PCR types do not have a name, but they are referred to by listing the pair (parent_record_type_name, child_record_type_name). Child record types of a specific parent are ordered.

A *hierarchical schema* (hierarchy) consists of number of record types and PCR types. Hierarchical schemas group into *hierarchical database schema*. In a hierarchy one record type is chosen as a *root*, and it does not participate in any PCR type as a child record type. Every

other record type participates as a child record type in exactly one PCR type. Any record type can be a parent record type in zero or more PCR types. If a record type does not participate as a parent record type in any PCR types then it is called a *leaf*. This description corresponds to that of a tree-structure. In the tree data structures terminology a record type would be a *node*, whereas the PCR type would be an *edge* of the tree.

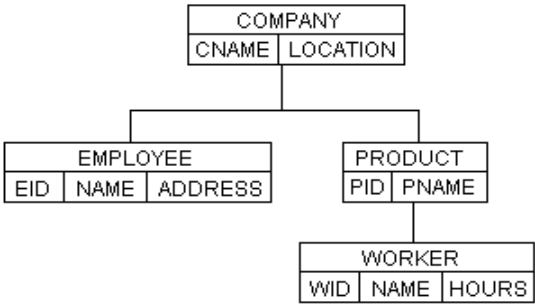


Figure 3.1.2. A simple hierarchical schema with two PCR types: (COMPANY, EMPLOYEE), (COMPANY, PRODUCT)

Hierarchical schemas have their hierarchical occurrences (also called trees or tree occurrences), each of them contains occurrences of child record types and one occurrence of a root record. Example is shown on a figure 3.1.3.

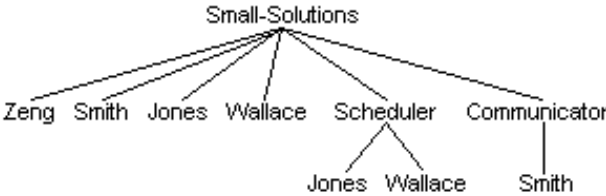


Figure 3.1.3. A hierarchical occurrence of the schema in Figure 3.1.2.

3.1.2. Virtual Parent-Child Relationship

In many cases 1:N relationship is not sufficient enough to represent for example a company organization. It often happens that a company develops more than one product, and one employee works on one or more products. In hierarchical data model it can be solved by allowing duplication of child record instances. But this solution, in addition to wasting storage space, may result in data inconsistency. IMS presented another solution to the problem of N:M relationships – the concept of virtual (or pointer) record type.

A *virtual (pointer) record type* VC is a record type with no data, but containing a logical pointer to a particular physical record. Instead of replication, we keep a single copy of a record and virtual records. Virtual record VC plays the role of a “virtual child” pointing to

a “virtual parent” in a “virtual parent-child relationship” (VPCR). Each record occurrence VC points to exactly one record occurrence of VP. IMS limits a record to being virtual parent of at most one VPCR, but each hierarchical database system provided its own solutions. In some implementations the hierarchical model became nearly network model, described later in this paper.

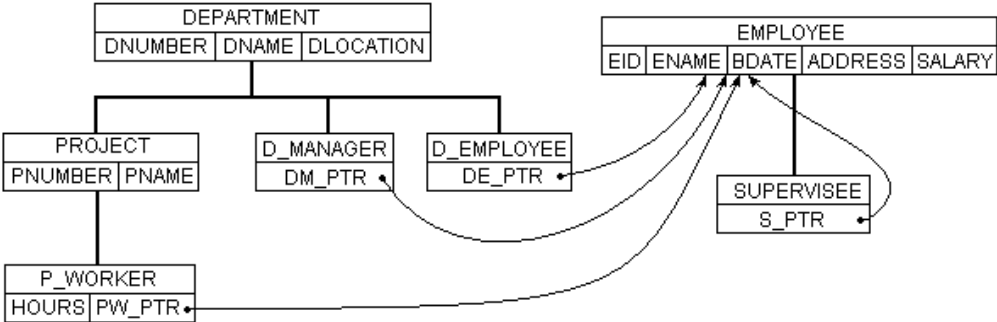


Figure 3.1.2. A hierarchical database schema with two hierarchies (first one with a DEPARTMENT as a root, and a second – with an EMPLOYEE) and VPCR types, to eliminate redundant record instances

3.1.3. DDL and DML in the Hierarchical Data Model.

Hierarchical Data Definition Language

Because there is no official standard of the hierarchical data model, each hierarchical DBMS implemented its own data definition language and data manipulation language. Instead of presenting any specific language, it will rather be shown how a hierarchical database schema could be defined using a meta-language. The example code below describes the database schema from the figure 3.1.2.

```

SCHEMA NAME := COMPANY
HIERARCHIES := {H1, H2}

RECORD
    NAME := DEPARTMENT
    TYPE := ROOT OF H1
    DATA ITEMS:= {
        DNUMBER      INTEGER,
        DNAME CHARACTER 15,
        DLOCATION     CHARACTER 30 }
    KEY:=DNUMBER
    ORDER BY DNAME

RECORD
    NAME := D_MANAGER
    TYPE:=CHILD OF DEPARTMENT
    CHILD NUMBER:=2
    
```

```

DATA ITEMS:= {
    DM_PTR POINTER
    WITH VIRTUAL PARENT:=EMPLOYEE }

RECORD
NAME := DEMPLOYEE
TYPE:=CHILD OF DEPARTMENT
CHILD NUMBER:=3
DATA ITEMS:= {
    DE_PTR POINTER
    WITH VIRTUAL PARENT:=EMPLOYEE }

RECORD
NAME := PROJECT
TYPE:=CHILD OF DEPARTMENT
CHILD NUMBER:=1
DATA ITEMS:= {
    PNUMBER    INTEGER,
    PNAME CHARACTER 20}
KEY:=PNUMBER
KEY:=PNAME
ORDER BY PNAME

RECORD
NAME := P_WORKER
TYPE:=CHILD OF PROJECT
CHILD NUMBER:=1
DATA ITEMS:= {
    HOURS REAL,
    PW_PTR POINTER
    WITH VIRTUAL PARENT:=EMPLOYEE }

RECORD
NAME := EMPLOYEE
TYPE := ROOT OF H2
DATA ITEMS:= {
    EID          CHARACTER 9,
    ENAME CHARACTER 35,
    BDATE        DATE,
    ADDRESS      CHARACTER 30,
    SALARYREAL }
KEY:=EID
ORDER BY ENAME

RECORD
NAME := SUPERVISOR
TYPE:=CHILD OF EMPLOYEE
CHILD NUMBER:=1
DATA ITEMS:= {
    S_PTR    POINTER
    WITH VIRTUAL PARENT:=EMPLOYEE }

```

To define a hierarchical database schema we must define the names of the hierarchies, record types, fields of each record type, and a data type for each field. If a field is a pointer we need

to assign a VP to it. For each record type we must also specify whether it is a child of another record type, or whether it is a root in one of the hierarchies. This is done in the TYPE clause. Data items defined under the KEY clause are constrained to have unique values for each record. Each KEY clause specifies a separate key. If more than one field is listed by a KEY clause, then the combination of these fields must be unique. CHILD NUMBER clause specifies left-to-right order of a child record type under its parent record type. Order of the individual records of a specific record type is provided by a ORDER BY clause.

Hierarchical Data Manipulation Language

HDML is a record-at-a-time language for manipulating hierarchical databases. It must be embedded in a general-purpose programming language called the **host language**. Most HDBMS software used PL/I or COBOL as the host language for its HDML.

In a record-at-a-time DML retrieval operations use program variables to store retrieved data. These variables exist in a part of a database system called the *user work area* (UWA). The last record accessed is usually called *current database record* (CR) and a pointer to it is stored in UWA. UWA also holds record templates for each record type defined in the database schema, and currency pointers for each hierarchy, pointing to the last accessed node in that particular tree type.

HDML commands can be categorized into three groups: retrieval, update and currency retention commands.

1. Retrieval:

```
GET (FIRST|NEXT) <record_name> [WHERE <condition>]
GET NEXT <child_record_name> WITHIN [VIRTUAL] PARENT <parent_record_name> [WHERE
    <condition>]
GET (FIRST|NEXT) PATH <hierarchical_path> [WHERE <condition>]
```

GET command retrieves a record into the corresponding program variable and makes it the current record. The simplest version of this command: GET FIRST always starts searching the database from the beginning of the hierarchical sequence until it finds the first record occurrence matching the given record name and conditions (if given).

GET NEXT finds the next occurrence of a specified record starting from a specific currency pointer. In IMS it is the CR pointer, whereas in System 2000 it was the pointer of a specific record type.

WITHIN PARENT clause narrows the search to the sub tree with a root being the indicated (by the current record pointer) parent record.

In a GET PATH command <hierarchical_path> is a list of record types that starts from the root along a path in a hierarchical schema, and <condition> is a Boolean expression specifying conditions on the individual record types along the path. The names of the records in the <condition> should be given precise. In general GET PATH command should retrieve all records along the specified path into the UWA variables, but in many implementations (IMS included) only some of the records can be retrieved. The example below describes a query “find first department, project and worker, given the condition that the worker spent more that 80 hours on the project”:

```
GET FIRST PATH DEPARTMENT, PROJECT, P_WORKER
WHERE P_WORKER.HOURS>80
```

2. Currency retention:

```
GET HOLD (FIRST|NEXT) <record_name> [WHERE <condition>]
```

Acts similarly to GET command, but the HOLD clause informs the DBMS that the program will update or delete the record just retrieved.

3. Update:

```
INSERT <record_name> [WHERE <condition>]
```

```
DELETE <record_name>
```

```
REPLACE <record_name>
```

The INSERT command inserts a record from a template variable into the database. If the record type is a root record of some hierarchy, then a new hierarchical occurrence tree is created with the new record as root. To insert a child record we need to make a chosen parent record the current record of the hierarchical schema before issuing the INSERT command, or specify the parent record in the <condition> clause.

To delete a record from we first make it the current record using GET HOLD command, and then issue the DELETE command. It must be noted here that deleting a node causes that the sub-tree with selected node as a root will be also deleted due to restrictions of the hierarchical model – only root records may exist in a database without a parent record. Also each VC record pointing to the record being deleted, should be deleted as well.

To modify a desired record we must make it the current database record with the GET HOLD command. Then we need to modify the desired fields in the UWA variable and issue the REPLACE command.

3.1.4. Summary

A hierarchical database structure is basically a *forest* data structure with records acting as nodes of trees. It may be represented also as a simple tree by adding a dummy record as a main root node. This model of data was very popular in the 1960s until the introduction of the CODASYL network model in 1970s. Although it is very hard to represent N:M relationships within the restrictions of this model, it still has its applications. An example of the hierarchical model in use today is MS Windows® registry or XML documents.

3.2. Network Data Model

The first database-standard specification, called the CODASYL DBTG 1971 report, was written in the late 1960's by the Database Task Group (DBTG) – a part of a committee CODASYL (Conference on Data Systems Languages). The standard presented is often referred to as the DBTG Network Data Model or the CODASYL Network Data Model. There have been many variations to the standard, implemented in the most popular DBMSs of the 70's, but in this paper only the objectives of the CODASYL data model will be presented.

3.2.1. Basic Concepts

A network database consists of two basic data structures – records and sets. *Records* are the means of storing data. Each record is a collection of fields (attributes), and each field has its name and a data type (format). Records are classified into *record types*. Record types describe the structure of a group of records, that store the same type of information. Each one has a name.

STUDENT			
NAME	ID	ADDRESS	MAJORDEPT

<u>Data item name</u>	<u>format</u>
NAME	CHARACTER 20
ID	CHARACTER 5
ADDRESS	CHARACTER 15
MAJORDEPT	CHARACTER 5

Figure 3.2.1. A record type STUDENT

Network model allows more complex data to be defined – a vector and a repeating group. A *vector* is a data item that may have multiple values in a single record. A *repeating group* allows inclusion of a set of composite values for a data item in single record.

STUDENT					
NAME	ID	ADDRESS	TRANSCRIPT		
			SEMESTER	COURSE	GRADE
Smith	...		I	Algebra	4
			I	Maths	4
			I	BHP	5

Figure 3.2.2. A repeating group TRANSCRIPT

Relationships between two record types are stored in structures called *DBTG set types*. Each set type consists of three basic elements:

- a name for the set type.
- an *owner* record type
- a *member* record type.

Each set type can have any number of *set occurrences* – actual instances of linked records. Each set instance consists of one owner record and any number of member records (even zero), which are ordered. Sets are customary shown as lines linking two boxes - records

Network data model allows only for one to many relationship. This implicates that a record from a member record type cannot exist in more than one set occurrence of a particular set type. However, a member record can participate simultaneously in several occurrences of different set types. (Figure 3.2.3)

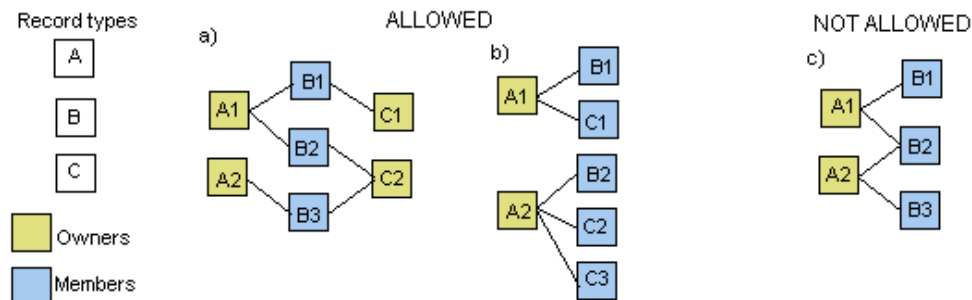


Figure 3.2.3. Correct record linking: a) 2 set types with 2 occurrences each, b) 2 occurrences of 1 multimember set type; Incorrect linking: c) record B2 has two owner in the same set type

CODASYL network model includes 2 special set types: System-owned sets and multimember set. Later after the original report was introduced, the third special set type has been added to the model – a recursive set. See figure 3.2.4 for visual comparison.

System-owned (Singular) set type has no owner record type, so the system is the owner. That is why there is only one set occurrence of it. The purpose of this set type is to provide entry points to the database via records of specific member type. Furthermore it can be used to order the records of a specific record type.

In the *multimember* set type, member records may be of more than one record type. The constraint that each member record may appear in at most one set occurrence is still valid, to enforce 1:N nature of the relationship. Multimember set types were not implemented in most of the popular commercial network database management systems.

In a *recursive* set type both owner and member records are of the same record type. It has been prohibited in the original DBTG report, because of difficulties in processing them using CODASYL data manipulation language (described in the section 3.2.3). For the same reason it hasn't been implemented into most of the network DBMSs of the 70s and 80s. ([1])

The problem of this type of relationship was customary solved by creating an additional linking (dummy) record type. Figure 3.2.4 c) shows the representation of the SUPERVISION relationship using two set types and a linking record type. We can think of the linking record as of an employee's role of a supervisor.

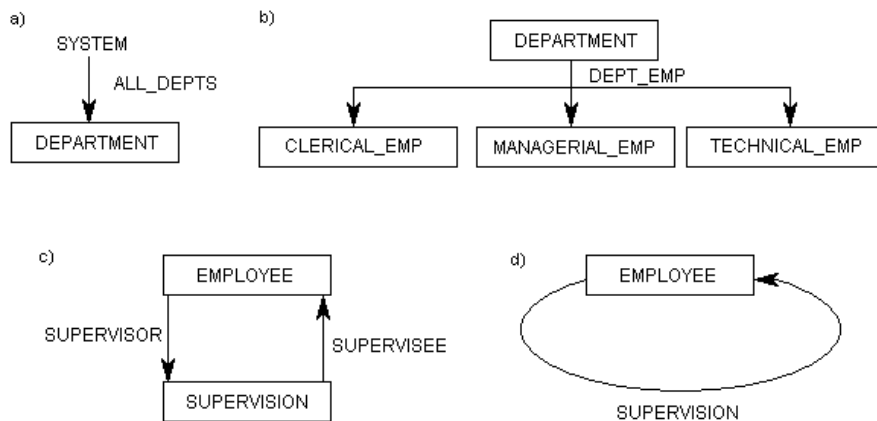


Figure 3.2.4. Special types of sets: a) system-owned; b) multimember set type; c) recursive set type SUPERVISION with linking record type SUPERVISION; d) prohibited recursive set type representation

Data manipulation language (more discussed in Section 3.2.3) is a record-at-a-time language. For each application program Network DBMS maintains a user work area (UWA), which is a buffer storage area that contains the following variables:

- record templates – a record for each record type accessed by the program
- currency pointers – a set of pointers to various database records most recently accessed by the program; they are of the following types:
 - o current of record type – one currency pointer for each record type, storing the most recently accessed record of specific type
 - o current of set type – similarly to the previous pointer type, but it must be noted that it may point to either owner or member record depending on whether an owner or a member was most recently accessed
 - o current of a run unit (CRU) – one single currency pointer containing the address of the record most recently accessed by the application program

3.2.2. Constraints on sets.

DBTG report mentions two types of constraints:

- insertion options
- retention options.

Insertion constraint on a membership specifies what is to happen when we insert a new record into the database, that is of a member record type. There are two options:

- Automatic – the new record is automatically connected to an appropriate set occurrence, as a member record. Choosing of a set is being made accordingly to the conditions in the specification of the set type. (See Section 3.2.3 for more details on specifying the conditions)
- Manual – the new record is not being connected to any of the set occurrences. The user may choose to do it, or to leave the record disconnected.

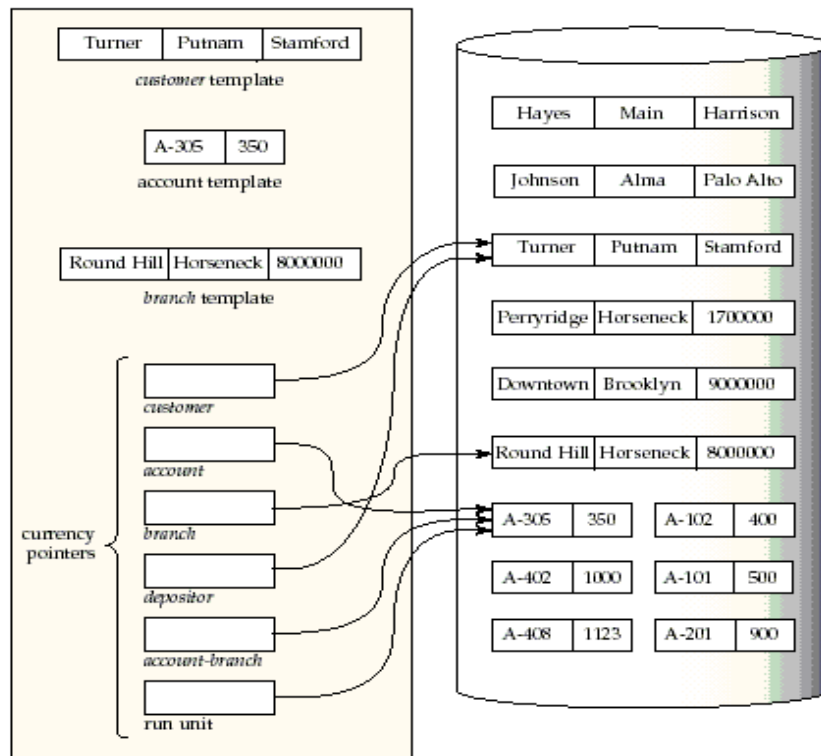


Figure 3.2.5. Example of the database and the UWA. Customer, account and branch are records. Depositor is a set with Customer as owner and Account as members. Account-branch has Account as owner and Branch as members.

Retention constraints describes the restrictions on disconnecting a member record from a set occurrence to which it is attached. There are three options:

- Optional – a member record can exist on its own, without being a member of any occurrence of the set type. It can be disconnected, reconnected and connected at will. If an owner record is to be deleted, all member records are disconnected, but remain in the database.
- Mandatory – a member record cannot exist on its own. Once it has been inserted into a particular set occurrence, it can only be reconnected to another occurrence of the same set type. It cannot be disconnected. Owner record cannot be erased until all the member records are reconnected to another set occurrence.
- Fixed – a member record cannot exist on its own. Once inserted into a particular set occurrence, it cannot be disconnected or reconnected. In order to reconnect it, it has to be erased, re-created and inserted into a new set instance. The member records are being deleted along with the deletion of the owner record

Insertion and retention options can be combined, and the report makes no restrictions on the combinations, but mandatory-manual and fixed-manual are not very useful.

As was mentioned before, the member records are ordered in set. User can define options used to order records, choosing from:

- ordering based on chosen ordering fields (see Section 3.2.3 on more detail)
- controlled by the time sequence of insertion

- system default – defined by the programmer of the DBMS
- *first* or *last* in the set occurrence in the time of insertion
- *next* or *prior* – new record is inserted before the current indicated record, or just after it.

3.2.3. DDL and DML in the Network Data Model.

Each Network DBMS has a slightly different syntax and additional options included in its DDL and DML. They also may have different *host language*. In the original DBTG report COBOL has been used as a host language, and it has been also the most popular one. Another popular host language was PL/I, with Pascal being the third popular choosing. This paper presents the main concept rather than exact syntax of any real DDL or DML. Lets have a look at some of the examples of DDL, taken from [1]:

SCHEMA NAME IS COMPANY

RECORD NAME IS EMPLOYEE

DUPLICATES ARE NOT ALLOWED FOR DEPTNUM
 DUPLICATES ARE NOT ALLOWED FOR FNAME, LNAME
 FNAME TYPE IS CHARACTER 15
 LNAME TYPE IS CHARACTER 15
 ID TYPE IS CHARACTER 9
 DEPTNUM TYPE IS NUMERIC INTEGER

RECORD NAME IS WORKS_ON

DUPLICATES ARE NOT ALLOWED FOR PNUMBER, EID
 EID TYPE IS CHARACTER 9
 PNUM TYPE IS NUMERIC INTEGER
 HOURS TYPE IS NUMERIC(4,1)

RECORD NAME IS PROJECT

DUPLICATES ARE NOT ALLOWED FOR NUMBER
 DUPLICATES ARE NOT ALLOWED FOR NAME
 NAME TYPE IS CHARACTER 9
 NUM TYPE IS NUMERIC INTEGER
 DESC TYPE IS CHARACTER 40

SET NAME IS ALL_PROJECTS

OWNER IS SYSTEM
 ORDER IS SORTED BY DEFINED KEYS
 MEMBER IS PROJECT
 KEY IS NUM

SET NAME IS EMP_WORKS_ON

OWNER IS EMPLOYEE
 ORDER IS SYSTEM DEFAULT
 DUPLICATES ARE NOT ALLOWED
 MEMBER IS WORKS_ON
 INSERTION IS AUTOMATIC
 RETENTION IS FIXED

KEY IS PNUM
SET SELECTION IS STRUCTURAL ID IN EMPLOYEE = EID IN WORKS_ON

SET NAME IS PR_WORKS_ON
OWNER IS PROJECT
ORDER IS FIRST
DUPLICATES ARE NOT ALLOWED
MEMBER IS WORKS_ON
INSERTION IS AUTOMATIC
RETENTION IS FIXED
KEY IS EID
SET SELECTION IS STRUCTURAL NUM IN PROJECT = PNUM IN WORKS_ON

Each set or record is given name by the NAME IS clause. The insertion and retention options, described previously are specified for each set using the INSERTION IS and RETENTION IS clauses. If the insertion is automatic, then we have to specify how the system will select a set occurrence using SET SELECTION clause. We then have three options:

- SET SELECTION IS STRUCTURAL – We can specify set selection by choosing two fields, which values must match – one field from the owner record type and one from the member record type. Owner record type must have DUPLICATES ARE NOT ALLOWED so that a single owner record would be selected.
- SET SELECTION BY APPLICATION – the application program makes the desired occurrence the indicated one, before storing a new member record
- SET SELECTION BY VALUE OF <field_name> IN <record_type_name> - a third option is used to specify a field of the owner record type whose value must match that of the UWA.

ORDER IS clause is used, as described previously, to set the order of the member records.

Data manipulation language for the CODASYL model consists of 8 commands divided into 4 groups (based on the command function):

1. Retrieval:

GET <record_name> – retrieves the current of run unit (CRU) into the corresponding user work area (UWA) variable

2. Navigation

FIND – locates a record in the database and resets the currency indicators; always sets the CRU and also sets currency indicators of involved record types and set types.

Syntax:

FIND (ANY|DUPLICATE) <record_type_name> [USING <field_list>]
FIND (FIRST|NEXT|PRIOR|LAST) <record_type_name> WITHIN <set_type_name> [USING <field_name>]
FIND OWNER WITHIN <set_type_name>

“Find any” locates the first record of given type whose <field_list> value is the same as the value of corresponding UWA field values. “Find duplicate” command finds the next (or duplicate) record, starting from the current record, that satisfies the search.

Once we have established a current set occurrence we can use “find owner” command to locate the owner record of a current set occurrence. We use “find first”, “find next”, “find prior”, “find last” to locate the first, next, prior or last member record of the set instance, respectively.

3. Record Update

STORE <record_type_name> – stores the new record (from the appropriate variable in UWA) in the database and makes it the CRU

ERASE <record_type_name> – deletes from the database the record that is the CRU

MODIFY <record_type_name> – modifies some fields of the record that is the CRU.

The modification process should be performed in 4 steps:

1. Make the desired record the CRU
2. Retrieve the record into the corresponding UWA variable
3. Modify the desired fields in the UWA variable (using the host language)
4. Issue the modify command

4. Set Update

CONNECT <record_type_name> TO <set_type_name> – connects a member record of a particular type (stored in the proper currency pointer) to the set occurrence indicated by the corresponding UWA pointer

DISCONNECT <record_type_name> FROM <set_type_name> – removes a member record (indicated by a currency pointer) from a set instance, marked in the UWA

RECONNECT <record_type_name> TO <set_type_name> – acts similarly to combined Disconnect and Connect commands

3.2.4. Summary

A network database is a collection of records and DBTG sets. This data model, along with the hierarchical model, was popular in the 70s. There have been a number of major commercial network DBMS implementations, e.g.: IDS II of Honeywell, DMS II and DMS 1100 of UNISYS, or IMAGE of Hewlett-Packard. Most of them were developed following the DBTG 1971 report. All systems have their own languages slightly varying from the one described in this paper, but all of them follow the concept of UWA and currency indicators. There has been studies on encapsulating relational data model in the CODASYL network data model [2]. In 1983 IDMS/Relational has been introduced, but the “pure” relational model eventually entirely replaced the CODASYL model. It is often claimed that the relational model is the first one with the scientific roots, but this is not entirely true, because the network model was based on the mathematical graph theory. This is why it is often referred to as the Graph Model

3.3. Relational Data Model

The relational model was first introduced by Edgar Frank Codd in 1970 [12]. During the early 1970's several experimental database management systems, based on the principles outlined in Codd's paper, were developed. In the 1980's commercial systems based on the relational model became dominant on market. Nowadays there are hundreds of books, works about the relational model. It has become the most frequently discussed data model. Unfortunately this resulted in many, often contradict, views on what the relational model is. In this paper we will discuss Codd basic model, regardless to later, mostly commercial, development.

3.3.1. Basic Concepts

The original relational model described in [12] is based on a single data structure – a mathematical *relation*. A basic relational building block is a *domain* (data type), nowadays often abbreviated to a *type*. A n-ary relation is a finite subset of a Cartesian product of n (not necessarily distinct) domains. In other words, a n-ary relation is a set of *n-tuples* of domain values. Domains should consist of simple values.

Codd proposed that users should deal not with domain-ordered relations, but with relationships which are their domain-unordered counterparts. To accomplish this domains are replaced by attributes – ordered pairs of *attribute name* and type name. Codd also suggested that he end-users need not know more about any relationship than its name together with the names of its attributes. Grouped definitions of relationships form a database schema.

Usually a relation is represented by a table, where tuple acts as a row. Each column represents an attribute.

In a relation one or more attributes must be chosen as a *primary key*, to uniquely identify each tuple. For each relation there may be one or more unique keys, but only one of them, selected arbitrarily, is the primary key.

Elements of a relation may cross-reference other elements of the same or a different relation. Keys provide a user-oriented means of expressing it. In a relation R we call a *foreign key* an attribute (or attribute combination), that is not a primary key, but its elements are values of the primary key of some relation S (it may also be R).

Figure 3.3.1. shows a simple database containing information about students, projects and coordinators. In this example data is stored in three relations: Projects, Coordinators and Students. In relation Projects, attribute Project_Id may be used as a primary key, whereas C_id is a foreign key with values from relation Coordinators

PROJECTS			
PROJECT_ID	NAME	C_ID	START DATE
P1	RAPORTING TOOLS	2	2003-09-10
P2	YODA DB	1	2004-09-10
P3	OPTIMALISATION	2	2005-04-01

COORDINATORS	
C_ID	NAME
1	Cay Michael
2	Masterton Peter

STUDENTS			
S_ID	NAME	YEAR	PROJECT_ID
1	Adams Scott	IV	P1
2	Camise Darla	IV	P1
3	Smith Robert	V	P2
4	Jaeke Anna	V	P3
5	Johnson William	III	P2
6	Anderson Andrew	IV	P1

Figure 3.3.1. Simple relational database

3.3.2. Normal Form

Real life provides situations where a relation does not consist of simple values, but of relations. For example – a firm need complex information on employees, their job record, salary record and information on their children (see figure 3.3.2. for a schema). Codd introduced a procedure (which he called **normalization**) for eliminating relation-valued domains. In order to *normalize* the set of relations following steps must be conducted:

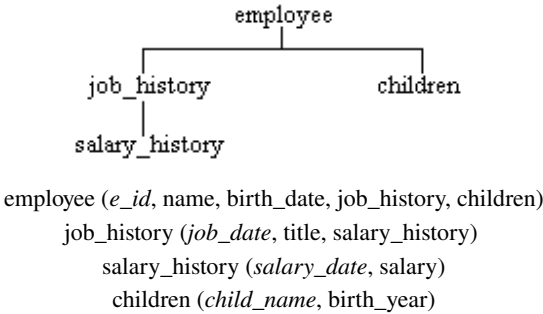


Figure 3.3.2. Unnormalized set

Starting with the relation at the top of the tree, its primary key must be taken and used for expanding each of the immediately subordinate relations by inserting this primary key domain or domains. The primary key of each expanded relation consists of the primary key before expansion augmented by the copied primary key. Next step is to strike out from the parent relation all non-simple domains, remove the top node of the tree, and repeat the procedure on

each remaining sub-node. The resulting collection of relations is called the first normal form (1NF). Figure 3.3.3 shows the result of normalizing the collection of relations from figure 3.3.2. The primary keys is written in italics.

employee' (*e_id*, name, birth_date)
job_history' (*e_id*, *job_date*, title)
salary_history' (*e_id*, *salary_date*, salary)
children' (*e_id*, *child_name*, birth_year)

Figure 3.3.3. Normalized set

With time additional normal forms were introduced by many authors. They were used to dispose of redundancy and improve performance of the relational database management systems (RDBMSs). Second and third normal form were introduced by Codd. All together there are 7 normal forms plus a non-first normal form (NFNF⁴), that (in opposition to the original 1NF) allows sets and sets of sets to be attribute domains. But in the original Codd paper only the 1NF was presented.

3.3.3. Operations on Relations and the SQL

Codd originally defined five relational operators: PERMUTATION, SELECT originally called RESTRICTION, PROJECTION, JOIN, COMPOSITION. Based on those operators, IBM research team tried to construct a query language, that would be suited to work with a RDBMS. In 1974 they presented SQL, which included DDL, DML and DCL in it. Nowadays there are hundreds of works on the SQL, and many various implementations of it. ANSI (American National Standards Institute) together with ISO (International Standards Organization) presented four standards of the SQL, each one with a bigger specification. In 1986 the first standard was presented, it was later ratified in the 1989, and that is why it is often referred to as SQL-1 or SQL-89. The next standard – SQL-2 – was presented in 1992, and in the 1999 – SQL-3. This standard was never implemented fully because of eclectics, internal contradictions and monstrous sizes of the specification. There also is a standard SQL – 2003, but it is not freely available⁵. It introduces XML-related features, *window functions*, standardized sequences and columns with auto-generated values (including identity-columns). It may be said that SQL is not a single language, but rather a family of languages. The

⁴ In literature NFNF is often written as NF²

⁵ It may be purchased from ANSI or ISO home page

difference between implementations is often seen by comparison of a way specific implementations deal with Null Values.

Because of this variety this paper will only describe few core aspects of the SQL irrespectively to any implementation.

1. Retrieval:

```
SELECT <list_of_record_names> FROM <table_name>
[WHERE <condition>]
```

SELECT command is used to retrieve zero or more rows from one or more tables in a database. The syntax shown above is only a basic syntax of a statement that can take very complicated forms. In most applications, it is the most commonly used DML command. In its basic form it directly corresponds to the *restriction* operation. The FROM clause is used to indicate from which tables the data is to be taken, and the WHERE clause is used to identify which rows to be retrieved. In the <list_of_record_names> we can specify the columns which are to be returned. It correspond to the *projection* operation but, in contradiction to it, SELECT command may return an array with duplicated rows. All of the mentioned before operations on relations have its corresponding SQL commands used within the SELECT clause. For example an additional clause to the SELECT statement – ORDER BY clause is an implementation of the *permutation*.

2. Data manipulation

```
INSERT INTO <table_name> [(<column1_name>, ..., <columnN_name>)]
    VALUES (<value1>, ..., <valueN>)
INSERT INTO <table_name> (<column1_name>, ..., <columnN_name>)
    SELECT <select_command_body>
```

```
DELETE FROM <table_name> WHERE <condition>
```

```
UPDATE <table_name> SET <list_of_assign_statements>
[WHERE <condition>]
```

INSERT is used to add zero or more rows to an existing table. The number of columns and values must be the same. If a column is not specified, the default value for the column is used. If a Select clause is used, the values it returns are inserted as new rows into the specified table.

DELETE removes zero or more existing rows from a table. Any rows that match the WHERE condition will be removed from the table. If the WHERE clause is omitted, all rows in the table are removed. The DELETE statement does not return any rows; that is, it will not generate a result set.

UPDATE is used to modify the values of a set of existing table rows. Either all rows in a table are altered, or only the ones that satisfy the condition defined in the WHERE clause. The assign statement is of a form: <Column_name> = <Value>

3. Data definition

DDL part of SQL allows the user to define new tables and associated elements. Most commercial SQL databases have proprietary extensions in their DDL, which allow control over nonstandard features of the database system. The most basic items of DDL are the CREATE and DROP commands. Some database systems also have an ALTER command, which permits the user to modify an existing object in various ways -- for example, adding a column to an existing table.

```
CREATE TABLE <table_name> (  
    <Column_definition_list>  
    [ PRIMARY KEY (<column_list> ) ]  
    [ <Foreign_key_definition_list> ] )
```

Column definition list atom:

```
<column_name> <column_type> [ <additional_options> ]
```

Foreign key definition list atom:

```
FOREIGN KEY <column_list>  
    REFERENCES <reference_table_name> (<ref_table_primary_key>)
```

```
ALTER TABLE <table_name> ADD <column_name> <column_type> <properties>
```

```
DROP TABLE <table_name>
```

CREATE causes an object (a table, for example) to be created within the database. On each column additional options and constraints may be added. For example: NOT NULL, DEFAULT <value>, UNIQUE.

ALTER TABLE command is usually used to add a column into a table, or to place constraints on it.

DROP causes an existing object within the database to be deleted, usually irretrievably.

4. Data control

DCL commands of SQL handle the authorization aspects of data and permits the user to control who has access to see or manipulate data within the database.

Its two main keywords are:

- GRANT — authorizes one or more users to perform an operation or a set of operations on an object.
- REVOKE — removes or restricts the capability of a user to perform an operation or a set of operations.

```
(GRANT|REVOKE) <Operation_names_list>  
ON (<table_name>|<schema_name>) TO <users_list>
```

Below is an example code that demonstrates the possible usage of SQL.

```
CREATE TABLE employee  
(e_id INT,  
name VARCHAR(45) NOT NULL,  
birth_date DATE,  
PRIMARY KEY (e_id)  
)
```

```
CREATE TABLE children  
(e_id INT,  
child_name VARCHAR(45),  
birth_year INT DEFAULT 1998,  
PRIMARY KEY (e_id, child_name)  
FOREIGN KEY e_id REFERENCES employee(e_id)  
)
```

```
CREATE TABLE job_history  
(e_id INT,  
job_date DATE,  
title VARCHAR,  
PRIMARY KEY (e_id, job_date)  
FOREIGN KEY e_id REFERENCES employee(e_id)  
)
```

```
CREATE TABLE salary_history  
(e_id INT,  
job_date DATE,  
salary_date DATE,  
salary REAL  
PRIMARY KEY (e_id)  
FOREIGN KEY e_id REFERENCES employee(e_id)  
)
```

```
INSERT INTO employee VALUES (1, 'Smith John', 1970-03-21)
INSERT INTO children (e_id, child_name) VALUES (1, 'Smith Susan')
```

```
SELECT name, birth_date FROM employee
WHERE e_id<10
```

3.3.3. Codd's 12 rules

As the benefits of the relational approach become more widely perceived, vendors of DBMSs increasingly often claim that their products are 'relational', not always with justification. Common belief was that it was sufficient to only *present* data in a set of tables. In 1985 Codd produced the following set of 'rules' by which systems should be judged:

1. Information Rule.

All information is represented at the logical level and in exactly one way - by values in tables.

2. Guaranteed Access Rule.

Each datum (atomic value) in a relational database is guaranteed to be logically accessible through a combination of table-name, primary key value and column-name.

3. Systematic Treatment of Null Values.

Null values (distinct from the empty character string or a string of blank characters, and distinct from zero or any other number) are supported in a fully relational DBMS for representing missing information and inapplicable information in a systematic way, independent of data type.

4. Dynamic On-line Catalog Based on the Relational Model.

The database description is represented at the logical level in the same way as ordinary data, so that authorized users can apply the same relational language to its interrogation as they apply to the regular data.

5. Comprehensive Data Sublanguage Rule.

A relational system may support several languages and various modes of terminal use (for example the fill-in-the-blanks mode). However there must be at least one language whose

statements are expressible through some well defined syntax as character strings, and that is comprehensive in supporting all the following items:

- Data definition
- View definition
- Data manipulation (interactive and by program)
- Integrity constraints
- Authorization
- Transaction boundaries (begin, commit and rollback)

6. View Updating Rule.

All theoretically-updateable views are also updateable by the system. (A view is theoretically updateable if there is a time-independent algorithm for unambiguously determining a single series of changes to the base tables, having as their effect precisely the requested changes in the view.)

7. High-level Insert, Update, and Delete.

The capability of handling a base table or a derived table as a single operand applies not only to retrieval of data but also to insertion, updating, and deletion. (This allows the system to optimize its execution sequence by determining the best access paths. It may be important in obtaining efficient handling of transactions across a distributed database, avoiding the communications costs of transmitting separate requests for each record obtained from remote sites.)

8. Physical Data Independence.

Application programs and terminal activities remain logically unimpaired whenever any changes are made in either storage representations or access methods. (There must be a clear distinction between logical and physical design levels.)

9. Logical Data Independence.

Application programs and terminal activities remain logically unimpaired when information-preserving changes of any kind that theoretically permit unimpairment are made to the base tables. (This rule permits logical database design to be changed dynamically, e.g. by splitting or joining base tables in ways which do not entail loss of information.)

10. Integrity Independence.

Integrity constraints must be definable in the relational data sub-language and storable in the catalogue, not in the applications program. Certain integrity constraints hold for every relational database, further application-specific rules may be added. The general rules relate to:

1. Entity integrity : no component of a primary key may have a null value.
2. Referential integrity : for each distinct non-null 'foreign key' value in the database, there must exist a matching primary key value from the same domain.

11. Distribution Independence.

A relational DBMS has distributional independence - i.e. if a distributed database is used it must be possible to execute all relational operations upon it without knowing or being constrained by the physical locations of data. This must apply both when distribution is originally introduced, and when data is redistributed.

12. Nonsubversion Rule.

A low-level (single record-at-a-time) language cannot be used to subvert or bypass the integrity rules and constraints expressed in the higher-level (multiple record-at-a-time) language.

Codd also added a 0 rule:

0. Foundation Rule.

A relational database management system must manage its stored data using only its relational capabilities.

Today none of the popular, commercial or not, database management systems satisfy all of those rules. But, nevertheless, they are being called "relational".

3.3.4. Summary

The relational model is still a dominant data model, although the research is being made on the next generation of data models – object model. For many years, because of relational model's scientific foundation, it has been seen as the best solution to the problem of storing and managing large amount of data.

It is ironic, however, that largely because of historical circumstances, its faithful implementations haven't yet succeeded in the marketplace. Early on, computers were thought not powerful enough to support it, and later on users got used to the shortcuts and the compatibility with previous implementations was a good enough excuse. While research in database theory built upon the foundation of the relational model, the DBMS industry has yet to faithfully implement the ideas that Codd laid out in the 70's.

As for the SQL language, there is a common belief that in the beginning it helped speed up the research on the databases, but in the end, because of its commercial success, halted the research for many years.

As for the popular claims that relational model is the only model with “clean mathematical roots” and it is more powerful than the network model, Albrecht Blaser and Herman Schutz, IBM researchers, in their paper “Data Base Research: A Survey” [11] published in 1975 wrote:

“(…) A first advantage of the graph model over CRM⁶ is the fact that only binary relations are used. This removes the need to distinguish between the domains via symbolic domain names. More importantly, due to its explicit notion of entities, which is not present in CRM, it is clean and clear in the mathematical sense without the need to deviate from it for "practical" reasons of convenience. Like for CRM, it is possible to develop calculus or algebra oriented language with the same rigor. On the other side it does not provide difficulty to furnish a user with a subview of the database (i.e. a subgraph) which restricts to the relationships, which may be seen by the user. (...) Since practically all known structures in computer science can be mapped conveniently to same form of graphs, it does not force us to exclude structures from our high level data modeling.”

⁶ Codd's Relational Model

3.4. Object-relational Data Model

During the 30 years after the introduction of the relational model many extensions of it were proposed. Many of them have been realized in commercial and non-commercial DBMSs. But what in this paper is called an object-relational (or post-relational) model is not exactly a data model, because there is no coherent theory on it. This paper presents five main extensions to the relational model.

Abstract data types (ADT)

The basic relational model allows attributes to be only of a simple data types. But it is not sufficient for applications like geographical information systems or computer aided design. They may need data types like polygons, documents. That is why some DBMSs give the user an abstract data type (ADT), which is a type of object that defines a value domain and a set of operations to work on those values. The details on an implementation of ADT is transparent for a user.

Nested relations

They are sometimes called NF² (non-first normal form) relations and are a proposition of handling complex objects. Those objects may be defined as objects with a hierarchical structure and with attributes that are atomic or also complex. Similarly to the plain relations, the nested ones have lists of column names. But a column name may refer to a group of attributes, and each of them may also refer to a group. This concept is similar to the concept of records in the network data model.

Patient history

P_ID	Name	Birth date	Consultation			
			Date	Time	Treatment	
					Code	Duration
2331	Johnson A.	1975-01-06	2006-02-01	09:00	1/XS	20
			2006-02-07	11:15	A/XS	35
6345	Smith J.	1969-11-09	2006-02-01	10:00	L/TP	50
			2006-02-02	10:00	L/TP	50
			2006-02-03	10:00	L/TP	50

Figure 3.1.3. Table with NFNF relations

To operate on those relations two internal operations are needed: *nesting* and *unnesting*. Nesting creates complex relation out of group of attributes, whereas unnesting is the opposite operation.

Triggers

Trigger is a procedural code that is automatically executed in response to certain events on a particular table in a database. Triggers can restrict access to specific data, perform logging, or audit access to data. There are two classes of triggers, they are either "row triggers" or "statement triggers". With row triggers you can define an action for every row of a table, while statement triggers occur only once and are not dependent on the shape of the data. Triggers cannot be separated from their table.

Stored procedures

Procedure is a procedural code that may be executed from a user program or another procedure. The exact implementation of a stored procedure varies from one database to another. In most cases however, stored procedures allow for an API to be defined for a database, rather than having a client application interact with the tables and other database objects directly. Stored procedures are supported by most major database vendors in some form, but there is debate amongst developers about the advantages of using stored procedures. Some people use them frequently, while others prefer to avoid using them at all.

Tables as objects

In some DBMSs (e.g. PostgreSQL) tables are treated as objects, attributes of tables are treated as attributes of objects, triggers and procedures as object methods. Nested relations also allow to keep all object attributes together, not separated by normalization. Tables may be joined into hierarchies with sub-tables inheriting attributes, binds, triggers and procedures from the parent table.

DBMSs using at least one of those extensions is said to contribute to the approach based on object (but not on the object approach). Many of the ideas of early object-relational database efforts have largely been added to SQL:1999. In fact, any product that adheres to the object-oriented aspects of SQL:1999 could be described as an object-relational database management product. For example, IBM's DB2, Oracle database, and Microsoft SQL Server, make claims to support this technology and do so with varying degrees of success.

3.5. ODMG's Object model

With the development of the computers it became more and more apparent that the basic relational model is not sufficient enough to store complex data, e.g. sounds, movies, or data that could easily be represented through graphs. The voices were heard that the relational model poorly represents the real world. In the OVUM Ltd. report from 1988 it was predicted that the databases based on an object model will soon become dominant on the market. In the 1991 a group called **ODMG** (Object Data Management Group) was formed. It's goal was to develop a coherent theory that would be a base for the object databases, just like the relational algebra was for the relational model and databases. First version of ODMG standard was created in 1993 , second version in 1997. Parallel to that research, in the 1980's prof. Kazimierz Subieta from the Polish Academy of Sciences started working on the Stack Based Approach (SBA) and on the Stack Based Query Language (SBQL) both will be described in the 4-th chapter of this paper.

3.5.1. ODMG manifest.

ODMG standard could be seen as a reaction to the attempt of creating a specification of obligatory feature of the object databases, published in 1990. OODBMS⁷ manifest proposes 13 obligatory futures for this systems. First eight rules show the meaning of compatibility of the OODBMS with the object rules. Last five rules determine that the OODBMS must handle a few features of the classical DBMS.

1. Handling of the complex objects. It must be possible to build complex object by the means of constructors for the basic objects such as set, tuple and list.
2. Handling of the objects identifiers. All objects must have unique identifiers, regardless of the values of those attributes.
3. Handling of the encapsulation. It requires the programmers to have access to the objects only through defined interfaces.
4. Handling of the object classes. A schema in OODBMS must contain a set of classes.
5. Handling of the inheritance of methods and attributes.

⁷ Object-Orientated Database Management Systems

6. Handling of the dynamic linking. To support the overwriting, DBMS must bind methods names with a logic during execution – it is called a dynamic linking.
7. DML must be computationally complete. A DML for the OODBMS should be a general purpose programming language
8. The set of the data types must be extendible. A user must have a capability to construct new types based on the predefined types.
9. The data persistence must be provided. Data must be stored after the shutdown the application, and user should not be forced to initiate persistence in direct way.
10. DBMS must be capable of managing very large databases. The OODBMS must have the mechanisms allowing for the efficient management of the memory (e.g. indices)
11. DBMS must provide synchronous access to the data.
12. DBMS must have the capability of recovery after a hardware or software failure
13. DBMS must provide an easy way of retrieving the data

3.5.2. Object Model

The object model (OM) defines the following modeling constructions: the basic elements are objects and literals. They are being classified by types. The state of the object is defined by its values or properties, which may be objects attributes or relations between objects. Database is described by the schema defined in the object definition language (ODL).

Objects

Objects are being defined based on the hierarchy types containing mainly abstract types: atomic objects, collective objects and structural objects. Each object has its identifier and may have a couple of names defined by a user.

Literals

Within the set of the literal types there is a null value, atomic literal, structural literal or collective literal. The value of the property of a literal cannot be changed and this is why literals may be used as plain data types. An example of the atomic literal is a string, whereas of the structural literal – date and time.

Collections

Collections may be objects or collective literals. The only difference between them is that the collective objects have identifiers. The object model defines 5 types of the collective objects:

- Set – unordered collection not allowing for duplicates
- Multiset - – unordered collection allowing for duplicates
- List –ordered collection allowing for duplicates
- Table – one-dimensional table of a varying length
- Dictionary – unordered sequence of keys and values not allowing for the duplication of keys

Types and Classes

The object model distinguishes types and classes. The type has only one specification and only one implementation. Each combination of specification and one implementation is called a class.

Properties

OM defines two new types of properties: attributes and relations. An attribute is defined on the one type of object and it has as a value literal or an objects identifier. Relations are defined between types and have a specific quantity

Operations

Instances of the object type are behaving in a way described in the set of operations, The object type definition defines the signature for the operation containing the name of the operation, its parameters, types for each parameter and a type of returned values. Operation may return an object as a value.

3.5.3. Object Query Language

Object Query Language (OQL) is a query language standard for object-oriented databases modeled after SQL. It does not contain any visible actualization operations, leaving it to the operations attached to each of the objects. It has been implemented in the O₂ Database system Below are the sample codes written in the O₂'s OQL:

```
SELECT e.child_name,  
       FROM e in Employees  
       WHERE e.salary>1000
```

```
SELECT j,  
       FROM Employee As e, e.job_history As j  
       WHERE e.name = "Smith"
```

```
SELECT struct(emp: e, child :c),  
       FROM e in Employees, e.child_name AS c
```

In the OQL structures must have labels. In the above example labels are “emp” and “child”. OQL places restrictions on the dot operator – it may only be used if the expression before it return only a single value.

OQL does not have command of a DDL, only DML ones.

For to use it in a database another language – for handling data definitions and control – must be implemented

3.5.4. Summary

The ODMG project has been closed in 2001, but the specification it released is not complete and there are voices that it is not implementable. Also the OQL language lacks completeness and commands that an object query language should possess. The main goal behind this project was to rebuild relational model theory to suite object-oriented approach. There have even been attempts to create object algebra. But all those attempts failed, and it became apparent that a completely new approach to the objectivity is needed.

4. Approaches to Query Languages.

The term “query languages” was introduced in the 1970’s. In the beginning it described user-friendly languages used by end-users of the database management systems. The first query languages: QUEL, QBE (Query-by-example) and SQL seemed to be ideal for an end-user, who is not a programmer. But this opinion changed with time. SQL and OQL, although trying to imitate natural language by using English words for commands, are complex and too difficult to a non-programmer person to use. Some commands are taken from programming languages and the syntax of more advanced commands is unclear. Below is an example query in SQL corresponding to the scheme from figure 3.3.2 from previous chapter. The goal is to obtain the names and birth years of children of employees that are employees for more than 6 years

```
SELECT child_name, birth_year FROM children
      WHERE e_id IN (SELECT e_id FROM employee emp
                   JOIN job_record job ON (emp.e_id = job.e_id)
                   WHERE job.job_date < 2000-01-01)
```

End-user nowadays instead of writing complex, multi-lined queries, prefer mouse – orientated self-explained interfaces with an easy, intuitive graphical choices. In this context “query language” means an intuitive tool designed for an end-user, used to browse through the data in a visual way.

“Query languages” are often perceived by mathematical theorists as a syntactic manifestation of theories like relational algebra or object algebra. However SQL, which is the main argument of this approach, is built only in 5% on the relational algebra, and the majority of its commands are taken from high – level programming languages (e.g. procedures, updates).

This leads to view that a query language should have capabilities of a programming language, or be embedded in one. There have been many tries to embed existing query languages (mainly SQL) in programming languages like Pascal, Java, or C. They only showed that it is a dead-end road because of incompatibilities between a query language and a programming language. The incompatibilities are collectively named “impedance mismatch”, which exposes some essential aesthetical and technical faults that are associated with the approach.

Another approach – creating a completely new query language that is also a programming language is worth consideration. It may be observed in ORACLE PL/SQL. Prof Subieta describes the following properties that a language of this kind should possess:

- **Conceptualization and abstraction** - the technical details (like file storage) should be of no interest to the programmer. A query language should support conceptual modeling of an application and should allow the programmer to access to databases on a very high abstraction level.
- **Declarativity** – a query language should allow the programmer to specify “what” needs to be done instead of “how” a machine should do it.
- **Macroscopic processing** – programmer should have the feeling that the part of some queries are processed parallel on any number of data.
- **Naturality** – a query language should be similar in use for a programmer as a natural language. But this does not mean constructing queries by using common words – it is not a solution. Rather it means that the query language should have a natural conceptual meaning for the programmer, and even the advanced queries should be written in a clear and readable form. Words used are of a little significance.
- **Efficiency** – the processing time of a query should be reasonable, and within programmers expectations. This means that a query language should involve advanced and powerful optimization methods.
- **Universality** – a query language should be able to support any retrieval goal that a programmer would wish, and retrieve the wanted data in a natural way, in a reasonable time.
- **Independence from an application domain** – the query language should not introduce notations that are not valid on all domains, with no exceptions.
- **Interpreting mode of execution, late binding** – all the building process should take place during run-time of an application, thus binding of names should be dynamic.

Prof. Subieta is also the inventor of the stack based approach (SBA) to the semantics and construction of the query languages. Under his guidance the basis of the Stack-Based Query Language were created. This chapter presents the chosen aspects of the SBA and the SBQL. The whole concept is described in [3]

4.1. Environment Stack

The concept of the **environment stack** (ENVS, also called a *call stack*) appeared in the 1950's when the first compilers of the high-level languages were created. Since then it has become the basis of many programming languages like Pascal, C, Java, Python... But it cannot be confused with the *result stack* (described below) – they serve different purposes.

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 environment 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 on 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.

The environment stack is not exactly a stack (following the definition from chapter 1.1.) It has also additional functionalities that concern name binding (which implies the search on the entire stack), scoping rule (skipping visiting some sections) and (in some cases) inserting new sections in the middle of the stack.

4.2. Name Binding

Prof. Subieta in [3] wrote that “binding is a compiler or run-time action that for a given name occurring in a source program subordinates a corresponding run-time entity, e.g. a main memory address, an object identifier, a start addresses of an executable procedure code, etc.”

Providing an environment stack stores environments together with all source names of run-time entities, a binding action for some name *X* is performed according to the following steps:

- The machine checks the top of the stack for an entity named X ; if there is such an entity, the binding returns it as the result and the action is terminated.
- If the top does not contain an entity named X , a section below the top is checked.
- Such a process is continued in lower and lower stack sections, till the entity named X is found. Visiting particular stack sections is governed by *scoping rules* that require omitting some sections;
- If X is not found on the stack, then the global environment is searched. The global environment contains static variables, database objects, computer environment variables, procedure libraries, etc. Alternatively, we can assume that the global environment is the lowest stack section - in such a case X must be found on the stack, otherwise an error should be reported.

For compiled languages like C or Java, binding mostly takes place during compilation (static or early binding) and rarely during run-time of a program (dynamic or late binding). An example of a static binding is a direct C function call: the function referenced by the identifier cannot change at runtime. An example of dynamic binding is dynamic dispatch, as in a C++ virtual method call. Since the specific type of a polymorphic object is not known before runtime (in general), the executed function is dynamically bound.

This leads to the concept of a binder, which does not occur in other approaches to query languages. Binder is also a key concept for query languages defined through SBA. It is a pair $n(x)$, where n is an external name which can be written in a source query or program, and x is a result of a query, in particular, a reference to an object or a value. Its role is to bind names, hence it joins an external name n with a run-time entity x that is denoted by this name. In the SBA the environment stack's section is a set of binders. Binding the name n means that we are looking at ENVs for a binder (binders) $n(x)$. The result of the binding is x . If the stack contains no such a binder, the situation can be qualified as a binding error.

4.3. Nesting

Nesting is an operation of opening a new section of ENVs with the interior of an object or procedure. In the SBA it is done with the help of the *nested* function. This function

takes any query result (as defined previously) as an argument and is implicitly parameterized by an object store. For the argument it returns a set of binders. This set is assumed to be pushed at the top of ENVs. The function *nested* is defined in [3] as follows:

- For a complex object $\langle i, n, \{ \langle i_1, n_1, \dots \rangle, \langle i_2, n_2, \dots \rangle, \dots, \langle i_k, n_k, \dots \rangle \} \rangle$ it holds: $nested(i) = \{ n_1(i_1), n_2(i_2), \dots, n_k(i_k) \}$. The case is illustrated in Fig.20. Indeed, *nested* returns the interior of the object identified by *i*.
- If *i* is an identifier of a pointer object $\langle i, n, i_1 \rangle$, and the object store contains the object $\langle i_1, n_1, \dots \rangle$, then $nested(i) = \{ n_1(i_1) \}$. The function for a pointer object returns the binder of the object that it points to.
- For a binder $n(x)$ holds: $nested(n(x)) = \{ n(x) \}$. For a binder *nested* returns this binder. As will be shown, this semantics is consistent with the typical understanding of auxiliary names introduced in queries.
- For a structure *nested* returns the union of the results of the *nested* function applied for elements of the structure: $nested(\mathbf{struct}\{ x_1, x_2, x_3, \dots \}) = nested(x_1) \cup nested(x_2) \cup nested(x_3) \cup \dots$
- 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.

4.4. Results Returned by SBQL Queries

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 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.

Below is the recursive definition of the Result set taken from [3]:

- Any atomic value stored in the database, or a value calculated by query operators, belongs to Result.

- Each reference to an object (object identifier) stored in the database belongs to Result. In particular, the domain Result includes also references to procedures, functions, views, methods and other behavioral entities if they will be introduced in a particular model of the object store.
- If $x \in \text{Result}$ and n is an external name, then the binder $n(x)$ belongs to Result. Such a result we will also call named value.
- If x_1, x_2, x_3, \dots belong to Result, then **struct**{ x_1, x_2, x_3, \dots } belongs to Result. *struct* is a structure constructor, which can be implemented as a flag decorating a result. The order of elements in a structure can be significant. Not all elements of structures must be named (i.e. elements need not to be binders). Also the situation where two elements have the same name is not excluded. A structure will be considered as a single (but composite) element, i.e. a structure is not a collection. As usual, types of x_1, x_2, x_3, \dots can be different. Implicitly, it is assumed that if a structure has a single element, **struct**{ x_1 }, then it is equivalent to this element x_1 . Structures having no elements are not allowed. This structure generalizes the concept of tuple, known from relational systems.
- If x_1, x_2, x_3, \dots belong to Result, then **bag**{ x_1, x_2, x_3, \dots } belongs to Result and **sequence**{ x_1, x_2, x_3, \dots } belongs to Result. *bag* and *sequence* are collection constructors, which can be implemented similarly as *struct*, by a flag decorating a query result. But a *struct* is considered to be a single element, whereas the *bag* and *sequence* are collections.
- The set Result has no other elements.

4.5. Query Result Stack

The **query result stack** (QRES, shortly: the *result stack*) is a stack necessary to accumulate temporary and final query results. Each QRES section contains an element of the Result set. It is an arithmetic stack known from the programming languages. In implementation QRES can be also used for other purposes like keeping counters of iteration loops during processing collections.

After evaluation of a query the top of QRES contains the final query result. This result is consumed by some agent within the application software, e.g. by a *print* command, by a

graphical user interface, by an updating or deleting clause, etc. Providing the application is free of inconsistencies, the beginning state of QRES is empty and the final state of QRES is empty too (all results of queries are consumed by the application software).

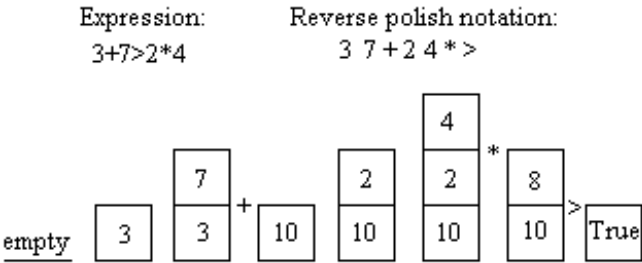


Figure 4.4.1. States of the arithmetic stack during evaluation of the given expression

4.6. M0 Model

In order to fully describe even an abstract language, definition of a structure of stored data is needed. The SBA is based on a family of data models: M0, M1, M2, and M3. In this paper only the M0 model is described, because it is sufficient to present the mechanisms of the SBQL. Description of other models can be found in [3].

In the M0 data model objects are represented as triples $o = \langle i, n, v \rangle$ where i is objects 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 already stored in database), than the object o is a **pointer object**
- If v is a set of objects than the object o is a complex object
- v may not be of any other type

In M0 the object store is a pair $\langle S, R \rangle$ where S is a set of objects and R is set of start identifiers. The set R determines entry points to the object store, that is, identifiers of such objects that can be starting point for searching and navigation by a query language. Each object in the store should be reached from objects having identifiers belonging to R ; that is, an object should be a sub-object of the object having an identifier in R (perhaps, recursively), or should be accessible by pointer objects.

The M0 model does not include types beside the general division into atomic, pointer and complex objects. There may be found similarities between M0 model and hierarchical model. Complex objects have a hierarchical structure, pointer objects can be seen as virtual relationships.⁸

```

S:
<i1, city_name, "Torun">
<i2, city_name, "Warsaw">
<i3, country_data, {<i4, name, "Poland">,
                    <i5, capital, i2>}>
<i6, person, { <i7, name, "Smith John">,
               <i8, lives_in, {<i9, city, i1>,
                              <i10, country, i3>} > }>
R = {i1, i2, i3, i6}

```

Figure 4.6.1. A simple database in M0 model

4.7. SBQL for the M0 Model

SBQL is a formalized object-orientated query language that may work with many data model, not only with M0. The base approach behind its creation was the SBA, whereas in SQL3 – nowadays leading query language supporting objectivity – it is a declarative approach, or as the critics state: “definition by example” approach.

The concrete syntax of SBQL is not fixed. Prof. Subieta reduced the SBQL’s syntactic considerations only to some abstract syntax. In concrete implementation any designer can invent an own concrete syntax that he/she considers the most adequate for the given purpose. So far the syntactic standard of SBQL does not exist. SBQL is considered as a theoretical frame or pattern for object-oriented languages, similar to the relational algebra or relational calculus.

In SBQL each atomic value v is called a **literal**, and is a part of a set L. Any name n belongs to a set N. Contrary to literals and names, identifiers cannot be subject of queries. The basic SBQL syntax (taken from [3]) is shown on the figure 4.7.1.

⁸ The other Mx models are corresponding more to the network model.

query ::= literal	The L set
query ::= name	The N set
query ::= unaryAlgOperator query	Unary algebraic operators
unaryAlgOperator ::= <i>count</i> <i>max</i> - <i>sqrt</i> not ...	
query ::= query binaryAlgOperator query	Binary algebraic operators
binaryAlgOperator ::= = < > + - * / and or ...	
query ::= query NonAlgOperator query	Non-algebraic operators
NonAlgOperator ::= where . join \exists \forall	
query ::= \forall query query \exists query query	Alternative (traditional) syntax for quantifiers
query ::= query as name	Name definition
query ::= query group as name	Grouping and name definition
query ::= if query then query	Conditional query
query ::= if query then query else query	Another conditional query
querySeq ::= query query, querySeq	Sequence of queries
query ::= struct (querySeq) (querySeq)	Structure constructor
query ::= bag () bag (querySeq)	Bag constructor
query ::= sequence () sequence (querySeq)	Sequence constructor

Figure 4.7.1. Basic SBQL syntax

SBQL divides operators into two groups: algebraic and non-algebraic operators. In this division many operators like join, projection, ordering, which in the relational algebra are treated as the algebraic operator, in SBQL they are non-algebraic ones. The non-algebraic operator are binary operators that modify the ENVS stack. Their additional feature is that during evaluation of the non-algebraic operators, firstly the left argument (query) is evaluated, and then for each element e from the result set of the left argument following three steps are performed: the ENVS is modified by nesting e , then the right query is evaluated, finally $pop()$ is performed on the ENVS.

Each operator that is not non-algebraic one falls under the category of algebraic operators (which do not modify the ENVS). The non-algebraic operators are for example: selection, projection/navigation, dependent/navigational join, quantifiers, ordering.

Unlike in SQL or OQL in SBQL there is no operator like **group by** and **null** values. Null values are generally avoided in the SBA. They are considered by K. Subieta as dangerous and are substituted by absent (optional) data. As for the *group by* operator it was considered unnecessary for object-oriented query languages, and substitutable by other operators

4.8. Comparison of SBQL, SQL and OQL Queries

Example 1:

Description	Get full information on students
SBQL query	Student
OQL query	Student
SQL query	Select * from Student

Example 2:

Description	Get the names and year numbers of all students
SBQL query	Student.(name, year)
OQL query	Select name, year from Student
SQL query	Select name, year from Student

Example 3:

Description	Get the names of all 1-st year students
SBQL query	(Student where year = 1).name
OQL query	Select name from Student where year = 1
SQL query	Select name from Student where year = 1

Example 4:

Description	Change the year number of a student named "Alan Granes"
SBQL query	(Student where name = "Alan Granes").year:=2
OQL query	Update Student set year:=2 where name = "Alan Granes"
SQL query	Update Student set year:=2 where name = "Alan Granes"

Example 5:

Description	Get the surname of the teacher of algebra (OQL omits the surname field !)
SBQL query	(Subject where name = "algebra").teacher.surname
OQL query	Select t from Subject as e, e.teacher as t where e.name = "algebra"
SQL query	Select * from Surname where t_id in (select t_id from Subject where name = "algebra")

Example 6:

Description	Is it true that each department employs an employee earning more than his/her boss? (example taken from [3])
SBQL query	\forall Dept (\exists employs.Emp (sal > boss.Emp.sal))
OQL query	for all d in Dept : exists e in select x from d.worksIn as x : e.sal > select y.sal from d.boss as y
SQL query	---- Not expressible ----

5. PySBQL

PySBQL is a query language based on the stack approach and Python programming language grammar, successfully implemented in the YODA system. This chapter describes the main concepts behind the parser and the interpreter of the PySBQL.

5.1. Main Concepts Behind the Language

During the implementation the following assumptions were taken into consideration:

- The syntax of the PySBQL will be based on the syntaxes of SBQL and Python.
- PySBQL will assume that the class definition is not obligatory but may exist as an object in the database, but no inheritance is supported.
- It will assume late binding, existence of functions, possibility of overloading the operators, declarative and imperative constructions.
- Temporary, permanent, individual and multiple data will be treated in the same way
- PySBQL must have the capabilities of the normal programming language
- It will work in the same way on any system platform.
- Indentation will be used for grouping statements (instead of curly brackets)

5.2. Why Python

Just like most of the query languages, Python is a dynamic, interpreted and interactive language. Here are the key features by which the Python language was chosen as a pattern for PySBQL:

- very clear, readable syntax
- strong introspection capabilities
- intuitive object orientation
- natural expression of procedural code
- very high level dynamic data types
- embeddable within applications as a scripting interface
- it is very easy to learn – so should be a query language

- very popular; thus it would be easier for a vast group of Python users to learn PySBQL
- it allows extension through modules – this feature implemented in PySBQL will enhance the programming abilities of PySBQL
- The Python 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 can be made
- it is different in usage from any other popular language thus PySBQL would present a new approach to the construction of query languages

5.3. PySBQL grammar

The complete grammar of PySBQL written in a manner similar to BNF notation is available in the file Sbql.jjt. This chapter will present the basis syntax of the language similarly to the way the syntax of SBQL language was presented in chapter 4.7. Additionally the table of operator priorities will be introduced.

PySBQL defines the literals set L and the names set N after the SBQL. Single lined strings are to be written within symbols: ‘a string’ or “a string”. Multi-lined strings are marked by symbols: ’’a multi-line string’’ or ‘’’’a multi-line string’’’’

query ::= literal	The L set
query ::= name	The N set
query ::= (<newline>)* query (<newline>)*	
BlockQ ::= <newline> <indent> (query)* <dedent>	Block of queries
query ::= (“ querySeq ”) [“ querySeq “]	
query ::= unaryAlgOperator query	Unary algebraic operators
unaryAlgOperator ::= + - ~ 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 assignOp query	
query ::= query . query	
query ::= query <- query	
assignOp ::= <- := += -= *= /= //= **= %= &= = ^= >>= <<=	
query ::= rename query as name	Name definition
query ::= create (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
query ::= for name in query : blockQ <newline> else : blockQ	
query ::= if query : blockQ (<newline> elif : blockQ)* [<newline> else : blockQ]	Conditional query
query ::= while query : blockQ [<newline> else : query]	
querySeq ::= query query, querySeq	Sequence of queries
query ::= “{“[query : query (, query : query)*]“}”	Dictionary creation
query ::= name ”(“ querySeq ”)”	Function call
query ::= break continue pass func classdef	
query ::= (return query) (delete query)	
func ::= def (name operator (bitOp arithOp)) “(“ [name = query (, name = query)*] “)” : blockQ	Function definition
classdef ::= class name : <newline><indent> (field func)* <dedent>	Class definition
field ::= (atomic pointer complex) field quantity	
Quantity ::= “[“ [0-9] .. ([0-9] *) “]”	

The below table presents the priorities of the selected operators. The operators later mentioned have lower priority:

Operator	Name
(...), [...], {...}, function()	Constructors and function call
x.y	Dot operator
+x, -x, ~x	Unary operators
x ** y	Power (right associative)
x * y, x / y, x // y, x % y	Multiplication, division, floor division, modulo
x + y, x - y	Addition, subtraction
x << y, x >> y	Bit shift
x & y	Bitwise and
x ^ y	Bitwise xor
x y	Bitwise or
x & y	Bitwise and
x < y, x = y, x in y, x is y, x is not y, x != y	Comparison
not x	Logical not
x and y	Logical and
x xor y	Logical xor
x or y	Logical or
x <+> y, x concat y	Concatenation
x where y	Selection
x group as y, x as y	Grouping
ref x as y, deref x	Reference and dereference

Operators not mentioned in the above table have the lowest priority.

5.4. Examples of PySBQL Queries and Statements

PySBQL allows for writing many types of statements. Unlike programming languages like Java, its blocks are denoted by indentation and not by symbols like { }. In many SQL implementations in order to complete functions body, or even to complete a query, a symbol ; must be inserted. In Java the same symbol is used to mark the end of expression. PySBQL, just like Python, uses new line symbols in place of ;. This results in a clear and readable form of the code. This chapter presents examples of queries and statements that are written in PySBQL. The words from the language are in bold

Example 1:

The chapter 4 starts with a sample code written in the SQL language. Its goal is to obtain the names and birth year of specific employees' children. The same query written in PySBQL has a form:

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

Example 2:

The following query might be written by a manager that would like to give each employee a raise by 100:

```
employee.salary+=100
```

Example 3:

In order to rename the field *town* in the *address* objects into *city* the following query may be executed:

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

Example 4:

The following query creates a new permanent (stored in the database) complex object *company* containing one atomic object named *name*, one pointer object *located_in* and one complex object *manager*

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

Example 5:

In order to print the names of all employees one of two following queries may be used:

```
print employee.name
```

or:

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

Example 6:

In order to count the money needed to pay out the salary for the employees and then print it onto a screen, a following query might be written

```
x:=0
for e in employee:
    x+=e.salary
print x
```

Example 7:

Suppose each person has a field *gender*. The following code adds to the name the social title Mr. or Mrs. accordingly to the content of the *gender* field

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

Example 8:

Suppose a user would want define a function to calculate the value of n! (PySBQL allows for short statements to be separated by the symbol ;) The function would be stored in the database as an complex object

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

The numbers printed on the screen would be: 120 and 1

5.5. Construction of the Parse Tree

The source code within the interpreter is represented by a binary tree structure of data. Tree is described by grammar and by one simple class which represents a node of a tree. This tree is created during the parsing of a source code. The parser is created using JavaCC (Java Compiler Creator) and JJTree – tools for Java programming language designed for creation of parsers. All the symbols (tokens) used in the PySBQL are defined as regular expressions in the beginning of the configuration file (PySBQL.jjt) based on which the parser was created. This file also contains all the terminals and non-terminals that create the context-free grammar, their priorities, and definition of the tree structure. Each node contains information on the type of terminal or non-terminal it represents, and the root node also contains a field to be used while identifying the user session from which it originated.

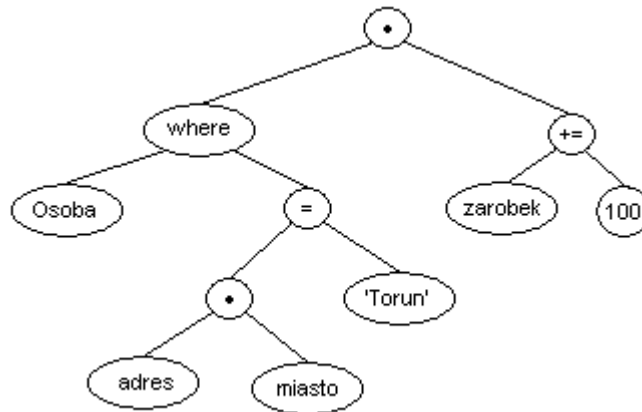


Figure 5.1. A sample parse tree created out of a query:
“(Osoba where adres.miasto= Torun).zarobek+=100”

5.6. Source Code Evaluations

The program is evaluated by going through the abstract syntax tree in the postfix order, and next for each of the nodes the evaluation procedure is started. The evaluation procedure is implemented using four classes grouped in one package. The main class Evaluator holds the method EvalTree(SimpleNode) which is responsible for evaluation of a given sub-tree. This solution allows to keep the code simple, with no unnecessary files and classes that would later on cause difficulties in the maintenance of the code.

5.7. Results of Expressions and the Result Stack

The class representing the set of results is called Result. It may store following values:

- atomic value (boolean, integer, double, string)
- reference to the object from the data storage
- dictionary structure
- list structure

The result of evaluating a sub-tree is placed on the result stack QRES and taken from it by the evaluator of the parent node. Every expression has its result and each result is being consumed by the evaluator of the parent node.

5.8. Environmental Stack

The environment stack has been implemented as a class Envs. It stores sections which are instances of a class Environ. The three lowest placed environments are static and available for each instances of Envs class. For each session a separate instance is being created. This solution allows for the support for multiple user and of transactions.

Environ class is used for storing binders – structures with their own names and values which are the same as of a class Result.

Envs class provides methods for retrieving binders, popping and pushing environments, and nesting object. The nesting method creates a new environment and fills it with the binder to the sub-objects and to the internal object of the appropriate class definition, if there is one.

5.9. Differences between SBQL and PySBQL

The main difference is that SBQL has three structures for holding collections of data – sequence, bag and struct whereas PySBQL has two: list and dictionary. List structure is somewhat similar to the bag structure – it may hold data of many types, and also lists and dictionaries. In the list the order of the content may matter, just like it may with sequence.

This solution was taken directly from the Python language which operates only on those two structure types for manipulating multiple data.

Another difference is made through adding to PySBQL Python operators of extended assignment and bitwise operators.

The imperative constructions were taken from Python, for example, the for loop has the form:

```
For <identifier> IN <Expression> :  
    <for_stmt_body>  
[Else: <else_body> ]
```

For every element from the result returned by <Expression> a loop will be executed. If there are no more elements for which to process a loop – the else statement (if existing) is called.

In PySBQL the statement for creating new complex object has the form:

```
create [temporary|local|permanent]<object_name>:(<sub_object>: <value>),...
```

In SBQL the variables are also objects. In PySBQL variables are simply binders. This solution was based on the need of assigning a list to a variable. This is not possible in SBQL. Also in PySBQL each object and value may represent Boolean values in a given context. Empty list, 0, False, "" (empty string) and atomic object, containing one of the previous, represent the **false** value. Each other value or object stands for **true** value

5.10. Differences between PySBQL and Python

The basic difference is that PySBQL is a hybrid of a programming and a query language and has full support for databases, whereas Python is a pure programming language. This is the reason for another difference: in PySBQL many binders of the same name are allowed in one environment, whereas Python restricts environments to at most one occurrence of binder of the same name.

Another aspect differing those two languages is the usage of symbol "=" for the comparison in PySBQL instead of "==" used in Python; and symbol "!=" instead of "=" for basic assignment. Also the "XOR" operator has been added – Python language already includes "^" (bitwise xor) but it lacked the *exclusive-or* operator for Boolean values.

In Python, if the binary operator has a list as one of the arguments and atomic value as the other throw an error (or if it is a "*" returns multiple concatenation of the list). PySBQL, in the same situation, returns a list created by performing the operation o each of the list elements. And this solution creates to another difference: in Python List1 + List2 returns a concatenation of both lists, whereas PySBQL returns a list of lists. For example:

In Python the result of: `[1,2,3] + [1,2]` would be `[1,2,3,1,2]`. In PySBQL the result is `[[2,3,4],[3,4,5]]`. For to concatenate 2 lists an operator `<+>` (or a key word **concat**) must be used. The last difference is that PySBQL does not support throwing exceptions and choosing a specific element from the list

5.11. Additional Modules

PySBQL has been created so that in future it would be easy to import modules in a Python-like way. Because of the vast similarity between Python and PySBQL it should be easy to rewrite Python modules into PySBQL modules thus enhancing the capabilities of the language. The command through the import of modules is performed is as follows:

```
import dotted (. name ) *
```

Also it is relatively easy to add other key words or extend the functionality of PySBQL by modifying two files: `Sbql.jjt` and `Evaluator.java`. The guidelines for this action are given in the chapter 1.

6. Content of the CD

Master Thesis text in two formats: pfd and sxw:

`/master_thesis/`

Eclipse workspace containing the implementation:

`/workspace/`

Source code folder:

`/workspace/src/`

Code created by the author of this master thesis:

`/workspace/src/code/evaluator/`

`/workspace/src/code/analyzer/`

`/workspace/src/code/Envs.java`

`/workspace/src/code/Environ.java`

`/workspace/src/code/objects/SBQLErrorException.java`

`/workspace/src/code/objects/SBQLEvalException.java`

`/workspace/src/code/objects/Result.java`

`/workspace/src/code/GConsole.java`

Class created partially by the author of this master thesis and by Michał Burzański

`/workspace/src/code/Binder.java`

7. Bibliography

1. Hopgood F.R.A. – „Metody kompilacji”, Państwowe Wydawnictwo Naukowe, Warszawa 1982 r.
2. Kościelski A.– „Teoria obliczeń” , Wydawnictwo Uniwersytetu Wrocławskiego, Wrocław 1997 r.
3. Subieta K. – „Teoria i konstrukcja obiektowych języków zapytań”, Wydawnictwo PJWSTK, Warszawa 2004 r.
4. Subieta K. – „A Critique of Object Algebras”, 1995, unpublished, may be found on the web page: <http://www.ipipan.waw.pl/~subieta/artykuly/CritiqObjAlg.html>
5. Waite W.M., Goos G. – „Konstrukcja kompilatorów”, Wydawnictwo Naukowo-Techniczne, Warszawa 1989 r.
6. Hopcroft J.E., Ullman J.D. – „Wprowadzenie do teorii automatów, języków i obliczeń”, Wydawnictwo Naukowe PWN, Warszawa 1994
7. <http://encyklopedia.pwn.pl/> – „wyraz”, „semiotyka”, „symantyka”, „składnia”, „gramatyka”, „semantyka”, „język”
8. R. Elmasri, S. B. Navathe – „Fundamentals of database systems”, Addison Wesley Publishing Company, 1994
9. www.wikipedia.org – “CODASYL”, “syntax”, “grammar”, “language”, “ODMG”
10. www.odmg.org – the main page of the ODMG group
11. A. Blaser, H .Schutz – “Data Base Research: A Survey”, LNCS, 1975
12. E.F. Codd – "A Relational Model of Data for Large Shared Data Banks", full text may be found on the web-page: <http://www.acm.org/classics/nov95/toc.html>
13. www.python.org – the main page for the Python programming language
14. www.jython.org – the main page for the Jython project – implementation of Python language written in Java.
15. A. Silberschatz, H. F. Korth, S. Sudarshan – “Database System Concepts”, McGraw-Hill Companies, 2001
16. C. J. Date – “Wprowadzenie do systemów baz danych”, Wydawnictwo Naukowo-Techniczne, Warszawa 2001 r.