A Persistent Object Store for the LOQIS Programming System*

Kazimierz Subieta

University of Hamburg
Department of Computer Science
Vogt-Kölln-Straße 30
D-2000 Hamburg 54, Germany †
e-mail: subieta@dbis1.informatik.uni-hamburg.de

Abstract

LOQIS is a high-level database programming system for personal computers, designed for the development of "intelligent" applications, such as teaching software, presentation systems and expert systems. It has many features of next-generation database systems, such as conceptual data view and processing, programming through queries, linked, complex objects, no static or dynamic limitations concerning the structure and size of objects, object identity, late binding, computational completeness, persistence, encapsulation, and inheritance. The paper presents the persistent object store (POS) that has been used as a basis for the implementation. In comparison to existing object stores, LOQIS POS deals not only with database objects but with all objects from the programming environment. This concerns programs in source and compiled versions, modules, types, schemata, arbitrary texts, graphic screens, diagnostic information, system messages, etc. All such objects are stored in POS on equal rights and are accessible by standard mechanisms. Arbitrary long fields can be stored as atomic values, an object can consist of arbitrary complex sub-objects, and each object can be dynamically updated, extended, or shortened with no limitations in structure or size. An object identifier can be used as a data value allowing arbitrary links among the objects. LOQIS POS works on two levels. The first is a kind of a heap which is used as a buffer for objects and for organizing dynamic structures (stacks, lists, etc.). The second level deals with persistent objects stored on a disc. Objects circulate between the heap and the secondary storage according to calls from the user program and/or from the environment. The circulation is fully automatic, i.e. to make space in the heap, objects are removed from it according to some policy. The system collects garbage automatically and on-the-fly. The package is implemented in C and works on IBM PC compatibles and on SUN.

Key Words

Persistent object store, object-oriented, object identity, query languages, object data model

*The paper is published in International Journal on Microcomputer Applications, USA, Vol.13, No.2, 1994, pp.50-61

†On leave from Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland

1 Introduction

LOQIS (Linked Objects in a Query Integrated System) [SMA90, Subi91] is a high-level programming system intended as a tool for the quick development of complex applications requiring non-traditional databases or knowledge bases. In our design assumptions we had in mind applications that require databases of moderate size, but of irregular, flexible and complex structure. Such databases occur in many fields such as educational systems, interactive presentational systems, office systems, expert systems, multimedia environments, and so on. The novelty of LOQIS relies on combining ideas arising in the database domain (such as data persistence, non-procedural query languages, conceptual data views, network data structures) with ideas from the domain of programming languages and programming environments. A description of LOQIS features follows.

- **Conceptual data view and conceptual data processing.** The application designer and programmer deal only with data that are seen through the conceptual view.

- **Support for complex objects**. LOQIS deals with complex data objects as closed units. Objects are hierarchies with an unlimited number of branches in each node and an unlimited number of hierarchy levels. The object and any of its components may be dynamically extended or shortened with no limitations. Atomic components of objects may store long values of unpredictable size, such as graphics or text.

- **Object orientation.** The concept *object-oriented database system* is understood differently. One extreme is presented in [ABD+89], where all attractive features of database and programming systems are considered to be attributes of object-oriented database systems. Other extremes are presented in [Ullm88], where object-orientation means object identity, complex objects and type hierarchy, and in [ZdMa90], where object identity, encapsulation and complex objects are left as sufficient attributes. LOQIS supports persistence, object identity, complex objects, a special form of inheritance, encapsulation, query processing, overloading, late binding, object sharing, etc., but because of the assumed hardware (PC), it does not support concurrency.

- **Uniform treatment of volatile and persistent data.** Data are stored on a disc and after a database is opened, all automatically belong to the program working area (without explicit reading commands). LOQIS does not differentiate between access to data stored in the database and access to volatile (temporary) data created during execution of programs.

- **Data querying through integration of associative and navigational access.** Relational query languages are based on the associative access, where the user specifies the properties of data to be found, while the navigational access means navigation in the database via pointers. It is sometimes claimed that associative access means *many-data-at-a-time*, while navigational access implies *one-data-at-a-time*. LOQIS integrates both styles following the *many-data-at-a-time* principle. The LOQIS query language has full computational power, including all kinds of joins, arithmetics, and transitive closures.

- **Full integration of the query language with programming capabilities.** Queries are used as parameters of procedural statements and as parameters of procedures. Output from functional procedures may be complex, i.e. it belongs to the same semantic domain as that for queries.

- **Integration with independent C packages**. Functions and procedures written in C can be integrated. They are linked dynamically with the program written in LOQIS.
- **Fully integrated programming environment.** LOQIS supports a programming environment, allowing many operations on persistent data objects and their components to be carried out. The environment includes all operations on programs such as editing, compiling, testing, and so on. Program modules are compiled independently and linked dynamically during execution. Programs are stored in the database as normal data; the same applies to diagnostic information, system messages, etc.

Implementation of the above LOQIS features required preparation of procedures isolating the programmer from details of physical data allocation and access. These procedures work on two levels. The first level, called *dynamic memory*, is a special heap with "elastic" main memory objects. It resembles a typical heap of C or PASCAL, but objects can be arbitrarily extended or shortened, and objects not used for some time may be automatically stored in a secondary file (this process is transparent for the programmer). Object identifiers are *handles* automatically generated by the system during object creation. A powerful set of stack-oriented and other operations on objects is implemented. Objects are considered to be uninterpreted sequences of bytes.

The second level, which we call *Persistent Object Store* (POS), deals with persistent objects stored on a disc. It uses dynamic memory procedures extensively. POS assumes a predefined internal structure of objects, and allows the organization of objects into complex hierarchies. Such complex objects can be manipulated as separate units. Each POS object has a unique persistent identifier, which can be used as a value of an attribute of another object. In this way POS objects can form an arbitrary network structure.

In comparison to many implemented persistent object stores, for example, Mneme [MoHo89, EMS88], the basic design assumption of LOQIS POS was that there be no static and dynamic limitations concerning the structure and size of objects. Lack of static limitations means the following properties:

- possibility to store inside objects arbitrary long (atomic) values, for example, graphic screen of the size of 0.5MB.
- possibility to create objects with an arbitrary number of attributes (perhaps, each of them may store an arbitrary long value).
- possibility to connect an object with other objects by arbitrary number of links. Links may lead not only to objects, but also to attributes, sub-attributes, etc.
- possibility to store repeating attributes, each of them with an arbitrary number of repetitions. An instance of a repeating attribute can be an arbitrary complex object.
- possibility to store an arbitrary object as an attribute of another object. This rule, applied recursively, means no limitation in the number of hierarchy levels for objects.

Real problems with implementing persistent object stores are connected with achieving proper dynamic properties. To the best of our knowledge, LOQIS POS gives the most powerful capabilities in this respect. The properties are the following:

- Each atomic value can be arbitrarily extended or shortened. If the value is shortened, the unused area is returned to the garbage on-the-fly.
- An arbitrary complex object can be inserted as a new attribute into any complex object.
- Objects, either atomic or complex, can be manipulated as separate units, that is, they can be deleted, inserted, moved, or copied as a whole by a single programmer's command.
- Links between objects can be removed, and arbitrary new links can be created.
- Instances of a repeating attribute can be removed, and new instances can be inserted.

Implementation of the above features requires negation of the concept that physical neighbourhood of data is a carrier of structural information. All structural information in LOQIS POS is organized by means of pointers rather than by the physical neighbourhood. The cost of this assumption is twofold: more storage space necessary and lower performance. Our tests carried out on two expert systems made in LOQIS have shown that extra storage space due to pointers is not larger that 10-20%, inessential for this kind of applications. Performance is a more critical issue. We considered and implemented many strategies to improve the performance; some of them are presented in the paper.

Persistent object stores have received a lot of attention recently, see [ACC83, ABL+91, Brow89, BrRo90, BDM87, BMM+91, CAC+84, EMS88, Guy87, KFC90, MoHo89, That90, VKC86]. The problem is not new. Many ideas are known from network (CODASYL) systems [CODA71, Bach74], from implementation of advanced relational database systems [SRH90, Ston87, KDG87], from implementation of programming languages [FiHa88], and so on. The term is usually associated with object-oriented systems [CDRS86, MaSt86, HoZd87, WoKi87, KBC+87, VBD89], and with the family of persistent strongly typed programming languages such as PS-Algol [CADA87], DBPL [ScMa91], Galileo [ACO85], Napier88 [MBCD89]. In these languages strong typing - to a great extent - forces a fixed format of data structures; unfixed formats are called *bulk data* and they present problems for these languages. Fixed formats of data structures have many advantages, in particular, they allow compact data representation and support performance. However, they have also well-known limitations. Many data from advanced applications have an inherently irregular format: variants, optional data, NULL values, repeating attributes, attributes of unpredictable size, very long fields of variable size (e.g. graphics), unpredictable number of links among data, etc., are examples of such situations. In contrast to fixed formats, irregular formats imply high requirements concerning object dynamics. Since LOQIS was designed for applications such as teaching software or expert systems, we start from the assumption that object dynamics must be supported, and that utilization of storage and performance are secondary issues. These assumptions determined further design decisions.

All functions of LOQIS POS are described in [Subi90].

The paper is organized as follows. In Section 2 we present basic LOQIS features. Section 3 describes the heap used for the implementation. In Section 4 we present details of POS and discuss problems of persistent object identifiers, semantics of the *delete* operator, and garbage collection.

2 Basic Features of LOQIS

2.1 Data Model

The LOQIS data model is described in [SuMi86, SuMi87]. Like other advanced data models, it was motivated by the limitations of the relational model with respect to conceptual modelling and object orientation. Basic formal assumptions of the model are the following.

Let I be the set of identifiers, N be the set of data names, and V be the set of atomic values. We do not assume any specific nature of V; in particular it may contain numerals, strings, texts from text editors, graphics, etc. Elements of this set are atomic, thus we never refer to their parts. Usually these values are subdivided into *domains* or *types* and associated with particular sets of operators, but at this stage these issues are inessential.

- *An atomic datum* can be considered either a triple $< i, n, v >$ or $< i_1, n, i_2 >$, where $i, i_1, i_2 \in I, n \in N$, and $v \in V$. The first triple represents a datum located under i, having a name n, and a value v. The second triple represents a pointer located at i_1, having name n and leading to the datum located at i_2. We assume $I \cap V = \emptyset$, i.e. there is an efficient method to distinguish atomic values and pointers.
- *A complex datum* is considered a triple $< i, n, \sigma >$, where σ is a set of atomic or complex data. Each triple in the complex datum has a unique first element. For example,

  $$< i5, EMP, \{< i51, NAME, Smith >, < i52, SAL, 2000 >, < i53, WORKS_IN, i6 >\} >$$

  is a complex datum.
- *A database instance* is a complex or atomic datum.

(Followers of object-orientation may change everywhere *datum* into *object*, and *identifier* into *object identity*.)

In this model we have ignored many important aspects of data structures, such as typing, classes, subdivision of data into modules, different variants of build-in bulk data types, permanence status of data, active data elements, and many others. Our model is extremely simple, but we show that it is powerful enough to cover important aspects of data structures.

Note that we do not require uniqueness of data names, on any level of data hierarchy. For example,

$$< i6, DEPT, \{< i61, DNAME, Toy >, < i62, LOC, Paris >, < i63, LOC, London >\} >$$

is a correct data structure with the repeating group LOC. A complex datum
$< i1, MyDatabase, \{$
$< i2, EMP, \{< i3, NAME, Brown >, < i4, SAL, 2500 >, < i5, WORKS_IN, i10 >$
$\} >,$
$< i6, EMP, \{< i7, NAME, Smith >, < i8, SAL, 2000 >, < i9, WORKS_IN, i14 >$
$\} >,$
$< i10, DEPT, \{< i11, DNAME, Toy >, < i12, LOC, Paris >, < i13, LOC, London >$

$\} >$,
$< i14, DEPT, \{< i15, DNAME, Sales >, < i16, LOC, Berlin >\} >\} >$
is also a correct data structure.

Hence we follow an unusual approach to *bulk data types*: they are considered collections of data (variables, in the programming languages terminology) having the same name. This understanding of bulk data types has advantages in comparison to typical understanding, mainly because of the *as-few-concepts-as-possible* principle (we avoid set and relation type constructors), and the *semantic relativism* which allows easier treatment of interaction of bulk data types with other data types. Our definition covers the tuple concept, but we do not introduce it in its own right. Notice that each tuple element has its own unique identifier.

We do assume neither a specific domain for data names nor special pragmatics. In particular, data names can be numbers. For example,

$$< i1, A, \{< i2, 1, Monday >, < i3, 2, Thusday >, .., < i8, 7, Sunday >\} >$$

in the terminology of programming languages is called an *array*. In comparison to other data structures, e.g. to records, arrays have an essential property: names of their elements can be calculated during run-time, i.e. the names are first-class objects. In LOQIS each data name can be calculated during run-time, thus the difference between arrays and other type constructors is neglected. Hence we are able to model data structures that include atomic data, records/tuples, arrays, sets, and arbitrary recursive combination of these constructors.

This model is directly implemented in LOQIS. Each datum, either atomic or non-atomic, has a name which need not be unique: any number of data with the same name may be created at each level of the data hierarchy. Figure 1 presents these features; DEPARTMENT, DEPT_NAME, EMPLOYEE, NAME, PREVIOUS_JOB, etc. are data names; "Sales" , "Jones", 4500, "engineer", etc. are atomic values.

Data may be connected by links that usually represent conceptual relationships between real-world entities. In Fig. 1 Jones and Levis are connected with the Sales department by the relationship WORKS_IN/EMPLOYS; this relationship is represented by links named WORKS_IN and EMPLOYS.

LOQIS allows the user to insert or to remove dynamically any data and allows the size of data to be changed dynamically. For example, see Fig.1, the programmer may remove JOB from EMPLOYEE, insert a new complex attribute PREVIOUS_JOB, assign a longer string to JOB, insert new links, and so on.There are no restrictions concerning the number of hierarchy levels of data structures; in particular atomic data X("I am a string") or Y(5) are correct data structures. They are analogous to atomic variables known from typical programming languages.

Considering the permanence of data the following cases can be distinguished:

- **permanent data** - a persistent component of the database;
- **temporary data** - created for a particular user-session and automatically cancelled after the session is terminated;
- **local data** - dynamically created inside a body of a procedure or function and automatically cancelled when it is terminated. As usual in programming languages supporting recursion, local data are located on a stack and are assigned to a particular call of a procedure/function.

Other properties of temporary and local data are the same as for permanent data; in particular, they may be complex and linked with other data, and there are no differences in access.

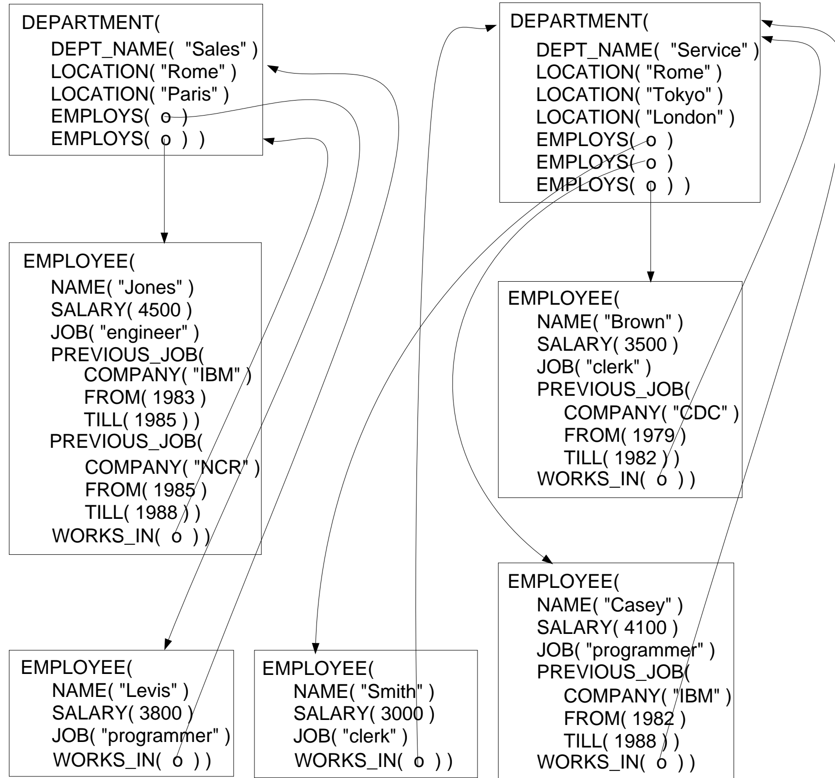

Figure 1: Example of data supported by LOQIS

2.2 Types and Data Description

LOQIS schemata are directly based on the concept of context-free grammars. That is, the description of a particular variable (or a set of variables) is a grammar determining all possible states of this variable (of the set). This approach can be considered alternative to well-known approaches to polymorphic languages [ADG+89]. Types are non-terminals in the grammar; recursive types are allowed, for example, the type *genealogical tree*, which is defined by itself. Currently we implemented a compiler of such schemata, which changes a schema into a finite automaton with a stack. This automaton is used as an efficient data checker. [1] Either a source schema and an automaton are stored in POS.

Figures 2 and 3 present a LOQIS schema. Two forms are illustrated: a diagrammatic entity-relationship schema for quick reference and a full schema written in LOQIS. Key word *repeating* denotes iteration, known from regular expressions. Alternatives are also possible; in particular, *optional* means an alternative with an empty string.

2.3 Expressions

Queries in LOQIS are called *expressions*. Indeed, all sentences such as '2*2', 'sqrt(X + Y)', 'EMPLOYEE where SALARY < 3000', etc. belong to the same

[1] Static type checking is not implemented yet; it implies non-trivial problems, such as testing if two languages defined by context-free grammars have a non-empty intersection.

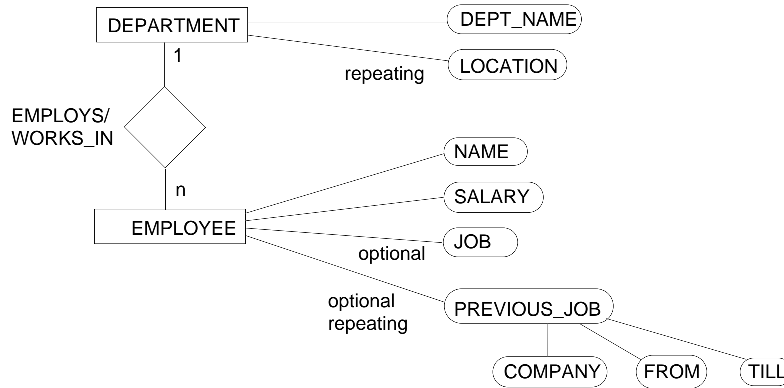

Figure 2: The diagrammatic schema

syntactic/semantic class.

After evaluation an expression returns a rectangular table consisting of atomic values and/or data identifiers. Identifiers returned by expressions may refer to complex objects, their attributes, sub-attributes, links, and so on. Identifiers are unprintable.

We explain principles of LOQIS expressions by examples (cf. Figs.1-3).

Example 1

```
EMPLOYEE where JOB = "programmer" and SALARY > 3000
```

The expression returns a one-column table with identifiers of EMPLOYEE entities having the attribute JOB storing the value "programmer" and with the attribute SALARY storing a value greater than 3000.

Example 2

```
EMPLOYEE where JOB = "programmer".
   ( NAME, (WORKS_IN.DEPARTMENT.DEPT_NAME), 0.3 * SALARY )
```

The expression returns a three-column table containing, for each programmer, the identifier of attribute NAME, the identifier of attribute DEPT_NAME of his/her department, and the number being 30% of his/her salary.

Example 3 Employees with salaries less than the average salary in Smith's department.

```
EMPLOYEE where SALARY < avg( DEPARTMENT where ("Smith" in
(EMPLOYS.EMPLOYEE.NAME)). EMPLOYS.EMPLOYEE.SALARY )
```

The operator '.' may be considered to be a generalization of the projection known from the relational model. It allows conceptual, many-path navigation in a network database. A many-step navigation is possible if attributes create a hierarchy, or if data are connected by links.

Example 4 Name and experience for each programmer working in the department employing Smith.

```
schema DEPT-EMP
   begin

      repeating entity DEPARTMENT
         begin
            DEPT_NAME( string )
            repeating LOCATION( string )
            repeating EMPLOYS( link to EMPLOYEE)
         end

      repeating entity EMPLOYEE
         begin
            NAME( string )
            SALARY( integer )
            optional JOB( string )
            optional repeating PREVIOUS_JOB( pjtype )
            WORKS_IN( link to DEPARTMENT )
         end

      type pjtype
         begin
            COMPANY( string )
            FROM( integer )
            TILL( integer )
         end

   end
```

Figure 3: A schema written in LOQIS

```
EMPLOYEE where NAME = "Smith". WORKS IN. DEPARTMENT. EMPLOYS.
EMPLOYEE where JOB = "programmer". ( NAME, sum( PREVIOUS_JOB.
( TILL - FROM + 1 ) )
```

LOQIS supports the full syntactic and semantic orthogonality of all operators. The following operators and constructs are available in LOQIS expressions:

- selections (operator *where*), navigation or projection (operator '.'), and navigational join (operator *with*);
- cartesian product (denoted ',') and set theoretic operators union (denoted '|'), minus and intersect;
- all standard comparisons of numeric values and comparison of strings (substring, superstring, etc.);
- set theoretic comparisons (equal, in, contains) and list comparisons (equallist, sublist, superlist);
- boolean operators *and, or, not*, arithmetic operators +, -, *, /, and string concatenation (denoted '&'),
- standard arithmetic and aggregate functions (abs, sign, sqrt, entier, ... count, min, max, sum, avg);
- boolean operator 'exists', testing presence of data of given name;
- function 'unique' removing duplicate rows from the argument table, and sorting operator 'order by';

- auxiliary names allowing temporary naming of parts of expressions (in the spirit of SQL correlation variables, but defined for more general cases);
- indirect names, calculated during run-time (e.g. indices of arrays);
- transitive closure operators (closed by, closed unique by, leaves by, leaves unique by) allowing processing recursive data structures and fixed-point calculations;
- calls of functions written in LOQIS, and functions from internal and external packages written in C. Since LOQIS functions are first-class objects (can be manipulated and stored in the database) and may return complex output, they are a direct generalization of the *view* concept, known from other query languages.

There are no problems with implementation of existential and universal quantifiers, but it appears that applicability of these operators is low, and they can be substituted by other operators (e.g. *exists*). Grouping was not implemented because of its non-orthogonality; it can be substituted by other operators. All types of joins are available through navigation according to links (which in network databases usually map referential integrities), the 'with' operator, and the cartesian product followed by a selection.

2.4 Procedural Constructs

LOQIS expressions may be used as arguments of procedural statements that act on a database. Since we use queries as arguments of commands, the commands perform quasi-parallel actions on many data. Both arguments of procedures and their output may represent complex values, e.g. relations. This is illustrated in Example 5.

Example 5 Function 'poor' has a list of jobs as a parameter. It returns names and salaries of employees of these jobs who earn less than the average, or the text "No poor employees".

```
function poor( JOBS )
  begin
    create AVERAGE( avg( EMPLOYEE.SALARY ) );
    (* Creating 0 or more pointers POOR pointing proper EMPLOYEEs *)
    create POOR( link to EMPLOYEE
                       where JOB in JOBS and SALARY < AVERAGE );
    if exists POOR then return POOR. EMPLOYEE. ( NAME, SALARY )
    else return ("No poor employee", 0)
  end;
```

Names and salaries of poor clerks and programmers:

```
poor("clerk" | "programmer" );
```

(End of example)

The following procedural constructs are implemented:

- assignment, insertion, insertion with copying, and delete (many-data-at-a-time, applied to atomic and complex objects);

- create new data (many-data-at-a-time), and change data name;
- *for each*, *while*, *loop*, and *case* statements;
- *if..then* and *if..then..else* statements;
- procedures (parameters being queries, functional procedures, *return* statement, etc.);
- utilities written in C (display, read, write, compile LOQIS procedure, etc.);
- calls of procedures from external packages written in C.

Arbitrary recursive procedures and recursive functional procedures are allowed. LOQIS supports procedures with procedural parameters.

2.5 The Programming Environment

LOQIS supports a complete programming environment which includes:

- creating, opening, closing, copying, renaming and deleting a database;
- editing, compiling, running, etc. LOQIS programs and schemata;
- showing, checking, renaming, altering, creating and deleting data; the system supports a browsing capability allowing navigation to any data in addition to the above operations;
- capabilities on data selected by queries: showing, renaming, and deleting;
- various other utilities: removing garbage areas from the database, testing and improving consistency of the database, etc.

2.6 Organization of Data in the LOQIS Programming Environment

The application written in LOQIS is subdivided into modules, as in Modula-2 and DBPL [ScMa91]. Modules are units of compilation, but in contrast to Modula-2 and DBPL, each module is a first-class object consisting of procedures, functional procedures, data description (types), data, and other information, e.g. determining export/import. Because of late and dynamic binding, each component of a module is a first-class object, i.e. it exists and can be manipulated during run-time. We do not implement nested modules (though there are no conceptual difficulties in the implementation). We introduced, however, the concept of the *main module*, storing some persistent data (common for all modules), and all other modules as its components.

Normally, local environments for procedure calls are considered to be volatile and stored on a stack. Since we assumed that properties of volatile variables should be the same as properties of persistent variables (this concerns their static and dynamic properties, and access mechanisms), it was impossible to employ a typical stack with fixed format for each variable. Thus we store such variables also in POS as a module; the stack contains only a pointer to this module. The module is cancelled when the procedure creating it is terminated. Local variables are assigned to persistent files, but because their life is usually short, they are almost never moved to them; they reside in the dynamic memory only.

An example of data organization for some point during the run of LOQIS application is presented in Fig. 4.

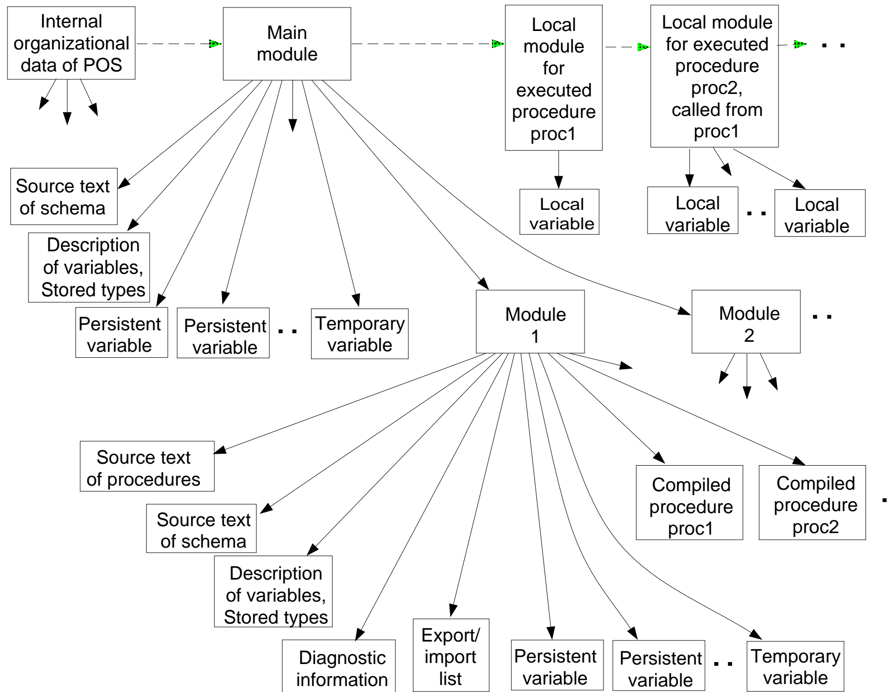

Figure 4: Data organization in LOQIS

3 Heap (Dynamic Memory)

The dynamic memory package consists of C procedures and functions which allow dynamic management of main memory objects of any size. The package functions are similar to typical heap functions with extensions concerning object dynamics. Each object is dynamically created and occupies an arbitrarily long, continuous part of the memory. Each created object may dynamically change its size, and the package is responsible for moving it, if necessary, to a free part of the memory, and for garbage collection. Objects are identified by unsigned numbers (2 bytes), returned by procedure DmCreate; this number is called the *handle* of an object. The idea is not new: dynamic heap objects, identified by handles, are also implemented in Microsoft Windows [Micro87]. Objects may be deleted at any time, and may have the size 0. Dynamic memory is supported by the powerful set of stack operations, allowing convenient organization of arbitrary stacks. Handles of objects can be stored inside other objects. In this way arbitrary hierarchies or other structures of objects can be created.

Objects not used for some time are stored, if necessary, in the secondary storage; this process is transparent for the programmer. The strategy of sending objects to the disc is based on counting access frequency: objects with lowest frequency are candidates for sending to the disc. Counting access frequency is done by a method with roots in *stochastic approximation*. Each object has a counter of access frequency, and there is a global counter of all accesses. At the beginning, each object obtains the average frequency (127). Each access to the object increases its frequency by 1 and increases the global counter by 1. When the counter is equal to the number of objects, it is zeroed and frequencies of all objects are reduced by 1.

To support interface to the C programming language, there are functions re-

turning pointers to objects. To achieve full consistency with C capabilities objects processed by C should be fixed. A fixed object cannot move in the memory and cannot change its size, thus may be processed by C as usual heap objects. Because of nested calls of C functions, which may process different aspects of the same objects, we associate with each object a counter of fixes rather than a single fixing flag. To achieve fixing/unfixing consistency, each fixing should be associated in a program with a twin unfixing. To unfix the object, the number of unfixes should be equal to the number of previously issued fixes. The maximal number of fixes (not cancelled by unfixes) is 255. A large number of fixed objects decrease the performance of the package or even may cause a crash due to memory overflow; thus objects should be unfixed as soon as possible.

The package maintains the index of all objects, which itself is an object stored in the dynamic memory. The index stores the following information about each object:

- Flags (primary/secondary, direct/undirect, etc.);
- Access frequency;
- The counter of fixes;
- Pointers to the object beginning and to the end;
- Handles of the previous and next objects (or NIL).

The last items form two double-linked lists, which reflect the order of objects in the main memory and in the secondary storage. They allow the package to retrieve the size of free areas before and after the object, and in general, enable efficient delete operations and garbage collection. For optimization, unfixed and short objects (up to 14 bytes) can be stored directly inside the index; in this case only the two first items describe the object. The handle of an object is simply a relative position of the object's description in the index, thus, given a handle, the pointer to the object description is calculated very quickly (by a simple arithmetic expression).

The dynamic memory package consists of the following functions:

- Create object, delete object, test if the object exists, put data into object, get data from the object;
- Stack operations: clear, push, pop, top, extract, distend (insert a free area), etc.;
- Pointer operations: fix, unfix, return pointer, test if a pointer points to the object interior;
- Miscellaneous: statistics, testing consistency.

4 Features of LOQIS POS

4.1 Basic Structures

The basic, elementary data structure is called an *atom*. An atom is an independent unit storing some elementary information, e.g. a record, an attribute of an entity, text, number, alpha-numeric screen, graphic screen, reference to another atom,

etc. Atoms may have different sizes (from 16 bytes to 16 Mbytes) and may be arbitrarily moved by the package through the storage space and main memory. Atoms may be independently cancelled, extended, shortened, or updated in any other way. The physical neighbourhood of atoms does not bear any information. Each atom is identified by a unique long (4 bytes) integer. These numbers are called *object identifiers* or *persistent pointers*; here for short they will be called *identifiers.* Sometimes an atom's identifier may be changed after updating (when the atom's size is increased), but updating of an atom never changes identifiers of other atoms. There are important reasons not to be totally orthodox with respect to persistent pointers; we discuss them later. If the atom changes its identifier after updating, its old identifier becomes a synonym of the new identifier. The programmer has a possibility to cancel old synonymous identifiers.

Atoms are linked into higher level structures. Two types of such structures, called *rings* and *spiders*, are introduced. A ring is a collection of atoms, where one atom is called *owner*, and the others are called *members.* An owner contains a pointer to the first member, the first member contains a pointer to the second, etc. The last member contains a pointer to the owner. A spider has a similar organization but its owner contains a directory to all members (the directory contains names and types of members), and each member has a pointer to the owner. A ring and a spider are illustrated in Fig. 5. Spiders allow faster processing at the cost of some extra storage space. From the functional point of view rings and spiders are equivalent: all functions of the POS package act on them exactly in the same way.

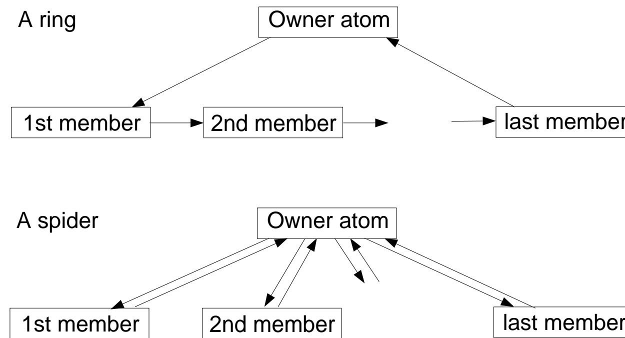

Figure 5: Basic POS structures

An owner is an atom of a special (*RING* or *SPIDER*) type. A degenerated ring/spider is possible, where the owner has no member. A special ring is the *main ring*, which contains only members and has no owner. A member atom in one ring/spider may also be an owner of another ring/spider. In this way an arbitrary hierarchy of rings/spiders may be organized. All POS atoms should be connected with the main ring, as its members, sub-members, sub-sub-members, etc. The number of atoms in a ring/spider is unlimited, as is the number of hierarchy levels of rings/spiders.

4.2 Links

To organize network data structures (necessary e.g. for implementing associative links or object sharing) cross-links in the structure are possible. Cross-links are organized by atoms of special types (of types *LINK*). A cross-link atom may be inserted as a member of any ring or spider, and it contains a reference (a pointer)

to another arbitrary atom. The package is responsible for full consistency of cross-links, i.e. when an atom's identifier is changed, all cross-links to it are automatically updated, and when the atom is deleted, all cross-links to it are also automatically removed. It is possible to define cross-links pointing cross-links.

Implementation of cross-links must enable navigation in the reverse direction. For several reasons, e.g. deletion, it should be possible to navigate efficiently from an atom to all link atoms that point it. This possibility is achieved by means of so-called *backward atoms* included after a referenced atom on usual principles for rings or spiders. Directly after the referenced atom A there is an atom containing pointers to LINK atoms that point the atom A. This is illustrated for the case of a ring in Fig.6. The package is responsible for consistency of backward atoms: inserting, updating or deleting of a link causes automatic action on backward atoms. Thus, backward atoms are transparent for the programmer.

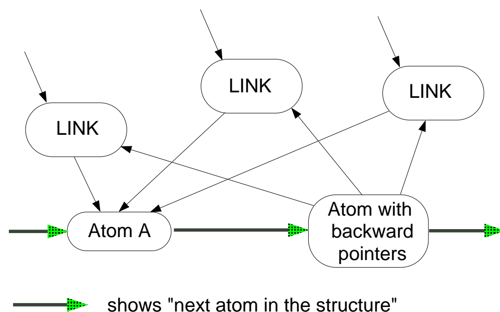

Figure 6: Organization of backward pointers

4.3 Persistent Object Identifiers

The problem of persistent identifiers (*object identities*) is frequently discussed in the literature, since there is no ideal solution; designers must choose some trade-offs. Basically, there are two approaches. The first, idealistic, assumes that object identities are always unique and each object - past, present and future - has an unique identifier, which is never changed. The disadvantages of this approach are the following: identifiers should be sufficiently long, to cover further demands for new identifiers, and the system must maintain a special table (*indirection table*) for conversion of identifiers into physical addresses of objects. The table would consume a lot of disc space, and would cause decreasing of performance, since an access to an object may be connected with a previous access to the table. Another solution assumes that object identifiers are somehow associated with physical addresses of objects. This approach is incompatible with the assumption that object identities are always unique, and makes problem when the object location must be changed; however, its big advantage is no the indirection table and good performance.

As in all systems, LOQIS POS makes a trade-off between these two extremes. Identifiers are in fact physical addresses, thus in general it is impossible to retain the atom's identifier in the case when its size is increased. This may present a difficult problem of updating auxiliary structures (for example, stacks) defined by the programmer if they store identifiers. To avoid this problem, a special atom type *PLACE HOLDER* is introduced. A place holder holds a place previously occupied by the atom (which size has been increased) and refers to the actual

atom's location; thus the identifier of the place holder works as a synonym of the atom identifier. An arbitrary chain of place holders is possible, i.e. when the atom size is increased a few times, place holders that point to place holders may also appear. The package keeps full consistency of place holders in the sense that their identifiers are equivalent to actual atom's identifiers and identifiers of place holders are never returned to a program that uses the package; thus place holders are transparent to the programmer. Place holders can be considered as a substitute of the indirection table, but they are created not for all objects, but only for those which sizes have been increased. Place holders consume some extra storage space (as usual for links, each place holder is associated with a pointer within a backward atom), thus they should be removed as soon as possible. Note also that because of the synonyms, comparison of identifiers for equality is more difficult and could be done by special options. Special functions are provided for removing place holders and for enabling updating auxiliary structures (created by the programmer) storing identifiers. All place holders are members of a ring and belong to the organizational data. An example situation with a place holder is presented in Fig. 7.

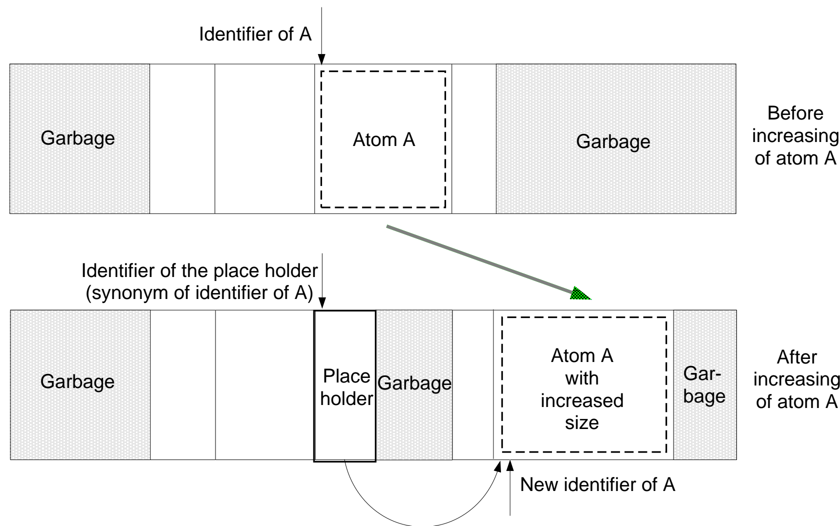

Figure 7: Inserting a place holder

4.4 Optimizations

In LOQIS POS we implemented many small optimizations, which were based on benchmarks in a real environment. In particular, we devoted some effort to develop a proper policy of object replacement in the dynamic memory. Current policy takes into account the size of atoms and statistic information (access frequencies). We feel, however, that much more can be won by involving semantic information about objects. This approach, for another environment, is proposed in [Ston91].

Below we present examples of optimizations. Frequently an atom should be included after the last member in a ring. To optimize this kind of operation a special atom type *TO END* is introduced. This atom is (optionally) included as a first member in a ring; this inclusion is automatically made by some functions. The atom points the last member in a ring. All public functions of the package do not see TO END atoms, hence they are transparent for the programmer.

Another optimization is connected with spiders. Formerly, they stored only pointers to members; but it appears that many operations need name and/or type of members. To avoid additional access to members, a table stored in a spider contains names, types and identifiers of all members.

Sometimes small decisions result in essential profits. For example, we decided that during allocation of new atoms we never leave a garbage area smaller than the minimal size of an atom. Surprisingly, this decision resulted in several percents gain in performance.

4.5 Atom's Structure

Each atom has a name, which need not be unique within the POS and need not be unique within a ring or a spider. Names, as strings of characters, are stored in a special name dictionary. A name is represented by a two-byte code. A 1:1 mapping between names and codes is assured. Some codes are reserved for special purposes; they have no counterparts in the name dictionary.

Byte	RING, LINK, PLACE HOLDER, TO END	LONG INTEGER	BYTES	SPIDER, BACKWARD
0	Type	Type	Type	Type
1	Flags	Flags	Flags	Flags
2-3	Code of atom's name	Code of name	Code of name	Code of name
4-7	Pointer to the next member in a ring, or to the owner	as before	as before	as before
8-11	Pointer to the first member in a ring, or to referenced atom	Integer value	Value size	Table size
12-15	Reserved for future use (zeroed)	as before	as before	as before
			Atom's value	Table of pointers and other data

The table of all types of atoms and total size of atoms of particular types is given above. In general, there may be many types RING, many types LINK, a single type BACKWARD, a single type PLACE HOLDER, a single type TO END, many types LONG INTEGER, many types BYTES, and many types SPIDER. Many types of RINGs or SPIDERs may be useful for distinguish their roles: for example, one role is a header of a module, another role is a header of repeating group, etc. The same concerns LONG INTEGER types: one role is normal integer, another to store time, another to store enumeration type, etc. LINK types may also have different roles, for example, one role is a link between persistent data, another role is storing export/import information among modules, another is storing relationship between data and their types, etc.

4.6 Organizational Data

Organizational data of POS are presented in Fig. 8. They form a complex datum which is stored in POS.

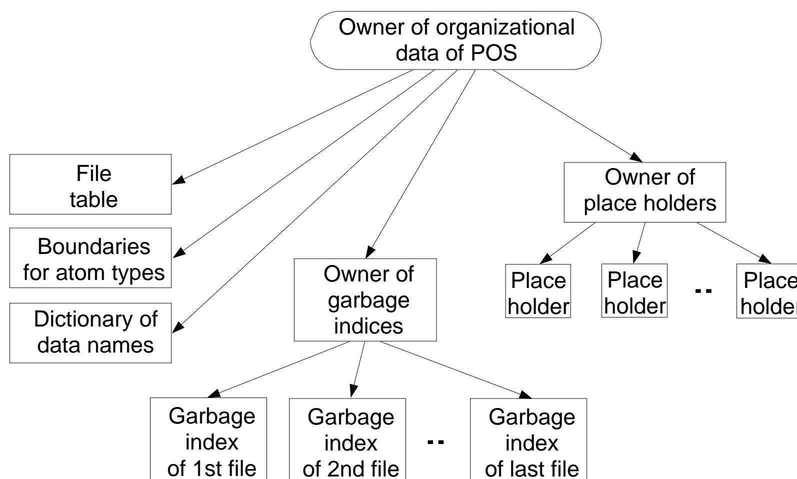

Figure 8: Organizational data

Organizational information include a directory of all atoms that are actually stored in dynamic memory. The directory is also a dynamic memory object. Each of its items consists of flags (updated/non-updated, full/head-only, etc.), atom's identifier, and the handle of a dynamic memory object storing the atom. The directory is organized as a hash-table according to atoms' identifiers; the overflow area is another dynamic memory object.

4.7 Functions of POS

We implemented functions for opening and closing POS, cancelling updates from the last checkpoint, creation and insertion of new atoms, retrieval functions, updating functions and various miscellaneous functions. Retrieval is performed by functions having usually names, types or identifiers as parameters and returning identifiers or other attributes of atoms. For example, AtmFind finds a first member with given name or type, AtmFindNext finds next such member, AtmGetValue copies atom's value to the determined area, AtmName retrieves atom's name, AtmNext and AtmOwner retrieve next member or owner respectively, etc. Updating functions permit all possibilities with respect to manipulation of the data structures assumed by POS package and consists of functions for assigning atomic and complex values, changing size of an atom, delete an atomic or complex object, assigning pointers, etc. All functions are consistent and transparent with respect to backward atoms, place holders, and TO END atoms. There are also functions on the name dictionary, functions for removing place holders, functions for statistics, testing consistency, garbage collection, and so on.

4.8 Delete and Garbage Collection

There are many approaches to garbage collection. Most of them are based on the principle of *lazy collection*: when an object is removed, only links to it are cancelled. The object exists until a special program (*collector*) returns it to garbage. There are many well-known algorithms following this approach, e.g. mark-scan, or reference count [FiHa88].

In LOQIS POS we do not follow the lazy approach. Each delete operation or shortening objects causes an immediate action on the garbage index, and index's items are immediately sticked if they describe neighbouring garbage areas. (Items of the garbage index describe all garbage area, even one-byte areas.) Thus our collector works on-the-fly: there is no special part of the program dealing with the collection. A basic reason for this decision is expected objects' dynamics. During the execution of an application atoms may be created and deleted very frequently. An example of such a situation is a procedure having local variables, which is called within a frequent loop. The variables are created when the procedure is called and deleted when it is terminated. Thus lazy garbage collection may cause unexpected growth of the number of objects in the dynamic memory, which will send them to secondary storage and, in effect, decrease performance. In general, for systems like LOQIS, which deal with explicit delete operations, we do not see an advantage in lazy garbage collection.

For deleting an atom we assume the following operations:

- deleting all links leading to the atom (this is easy to do because of backward pointers); deleting the last link causes removal of the backward atom;
- deleting (recursively) all structures subordinated to the atom;
- disconnecting the atom from the chain of members (updating properly pointers in the chain);
- deleting the atom itself.

In this way we never obtain an inconsistent structure (e.g. pointers leading to garbage).

A disadvantage in our method of garbage collection is fragmentation of the storage after a long period of work. This causes, from one side, unnecessary growth of files, and from another side, growth of garbage indices, thus lower performance. To overcome this problem we implemented a special utility for the reorganization of a database, which removes all garbage areas and place holders. This utility, obviously, changes identifiers of atoms, but leaves the structure consistent and isomorphic with the old structure. The utility recursively scans the database structure, as the mark-scan algorithm does, and rewrites atoms to the new file. In the result, physical atoms' order is compatible with the logical order, e.g. all attributes of an object are in one physical place. Sometimes it is claimed that such an organization supports performance, because a whole object is stored at one physical page or at neigbouring pages. Our tests have shown that such claims are unjustified. After such a reorganization the performance was decreased. There are two reasons for that. First, lack of garbage areas near atoms cause no possibility of their growth, thus increasing atom's size causes copying them to the end of a file (with allocating place holders, etc.). Second, together with frequently accessed attributes, a physical page may contain attributes with very low access frequency; these attributes are unnecessarily copied together to the main memory. Thus we considered (but have not yet implemented) a reorganization policy, which will take into account some *friendliness* between data, i.e. which store together such data, which are likely to be together called to the dynamic memory by an application program. Similar ideas are presented in [BDH91].

5 Conclusion

We presented, in some detail, a persistent object store that is implemented for the LOQIS database programming system. The package follows design assumptions of

languages implemented in LOQIS, and assumptions concerning the database and programming environment. Required static and dynamic properties of stored objects implied the necessity of quite new approaches to the implementation. The basic design assumption was negation of the concept that physical neighbourhood of data is a carrier of structural information: every complex structure is organized by means of pointers. At this stage the package works with a satisfactory performance for applications of moderate size; however, large databases may cause essential performance problems. Growth of organizational data due to large amount of pointers appears to be inessential, because in assumed applications (teaching software, expert systems, etc.) there are many large atomic objects, such as texts or graphics.

The most important topics for further development are: concurrency and transaction processing, and improving performance through indices and utilization of information on semantics of objects and their behaviour.

Acknowledgements

The author would like to thank the following people for their contribution to the project and this paper. Design assumptions of LOQIS POS were discussed with Marek Missala and Krzysztof Anacki, who participated in the LOQIS project. Florian Matthes and Andreas Rudloff made many helpful comments concerning this paper. Plamen Kiradjiew tested a version for SUN, and made suggestions on how to improve it. Prof. Joachim W. Schmidt arranged excellent conditions for my work. Helen Brodie helped me in preparing the paper and corrected my English.

References

[ACO85] A. Albano, L. Cardelli, R. Orsini. Galileo: A Strongly Typed, Interactive Conceptual Language. ACM Transactions on Database Systems, Vol.10, No.2, pp.230-260, 1985

[ADG+89] A.Albano, A.Dearle, G.Ghelli, C.Marlin, R.Morrison, R.Orsini, D.Stemple. A Framework for Comparing Type Systems for Database Programming Languages. Proceedings of the Second International Workshop on Database Programming Languages. Gleneden Beach, Oregon. Morgan Kaufmann Publishers, pp.170-178, 1989

[ACC83] M.P. Atkinson, K.J. Chisholm, W.P. Cockshott. CMS - A Chunk Management System. Software Practice and Experience, Vol.13, pp. 273-285, 1983

[ABD+89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, S. Zdonik. The object-oriented database system manifesto. Proc. DOOD 89, Kyoto, Japan, pp. 40-57, 1989.

[ABL+91] M. Atkinson, V. Benzaken, C. Lécluse, P. Philbrow, P. Richard. Experiments with Persistent Map Stores. ESPRIT BRA Project 3070, Report FIDE/91/22, 1991

[Bach74] C.W. Bachman. Implementation techniques for data structure sets. In: D.A. Jardine (ed.): Data Base Management Systems, North-Holland, pp. 147-157, 1974.

[BDH91] V. Benzaken, C. Delobel, G. Harrus. Clustering Strategies in the O_2 Object-Oriented Database System. ESPRIT BRA Project 3070, Report FIDE/91/21, 1991

[BDM87] A.L. Brown, A. Dearle, R. Morrison. An Architecture for a Strongly Typed Persistent Object Store. Proc. of Workshop on Object-Oriented Programming Systems, Languages and Applications, Orlando, Florida, 1987

[Brow89] A.L. Brown. (Ph.D. Thesis) Persistent Object Stores. Universities of Glasgow and St.Andrews. PPRR-71, Scotland, 1989

[BMM+91] A.L. Brown, G. Mainetto, F. Matthes, R. Mueller, D.J. McNally. An Open System Architecture for a Persistent Object Store. ESPRIT BRA Project 3070, Report FIDE/91/31, 1991

[BrRo90] A.L. Brown, J. Rosenberg. Persistent Object Stores: an Implementation Technique. University of St.Andrews, Dept. of Mathematics and Computational Science, Research Report CS/90/15, 1990.

[CDRS86] M.J. Carey, D.J. DeWitt, J.E. Richardson, E.J. Shekita. Object and File Management in the EXODUS Extensible Database System. Proc. 12th VLDB Conf., Kyoto, Japan, pp. 91-100, 1986.

[CAC+84] W.P. Cockshott, M.P. Atkinson, K.J. Chisholm, P.J. Bailey, R. Morrison. A Persistent Object Management System. Software Practice and Experience, Vol.14, No.1, pp.49-71, 1984

[CODA71] CODASYL Database Task Group Report, ACM, New York, 1971

[CADA87] R.L. Cooper, M.P. Atkinson,A. Dearle, D. Abderrahmane. Constructing Database Systems in a Persistent Environment. Proc. 13th VLDB Conf., Brighton, England pp. 117-125, 1987.

[EMS88] J. Eliot, B. Moss, S. Sinofsky. Managing Persistent Data with Mneme: Designing a Reliable, Shared Object Interface. Proc. of the OODBS II Workshop, Bad Munster, Germany, Sep. 1988, Springer-Verlag 1988.

[FiHa88] A.J. Field, P.G. Harrison. Functional Programming. Addison-Wesley 1988.

[Guy87] M.R. Guy. Persistent Store - Successor to Virtual Stores. Proc. of 2nd International Workshop on Persistent Object Stores, Appin, August 1987.

[HoZd87] M.F. Hornick, S.B. Zdonik. A Shared, Segmented Memory System for an Object-Oriented Database. ACM Transactions on Office Information Systems, Vol.5, No.1, 1987.

[KBC+87] W. Kim, J. Banerjee, H.-T. Chou, J.F. Garza, D. Woelk. Composite Object Support in an Object-Oriented Database System. Proc. OOPSLA Conf., Orlando, 1987

[KFC90] S. Khoshafian, M.J. Franklin, M.J. Carey. Storage Management for Persistent Complex Objects. Information Systems, Vol.15, No.3, pp. 303-320, 1990

[KDG87] K. Kuspert, P. Dadam, J. Gumauer. Cooperative Object Buffer Management in the Advanced Information Management Prototype. Proc. 13th VLDB Conf., Brighton, England pp.483-492, 1987.

[MaSt86] D. Maier, J. Stein. Indexing in an Object-Oriented DBMS. Proc. Int. Workshop on Object-Oriented Database Systems, Pacific Grove, 1986.

[Micro87] Microsoft Windows, Software Development Kit, Programmer's Learning Guide. Version 2.0. Microsoft Corporation 1987

[MBCD89] R. Morrison, A.L. Brown, R. Connor, A. Dearle. The Napier88 Reference Manual. Universities of Glasgow and St.Andrews PPRR-77, 1989

[MoHo89] J.E.B. Moss, T. Hosking. Managing Persistent Data with Mneme: User's Guide to the Client Interface. (Unpublished report), 1989

[ScMa91] J.W. Schmidt and F. Matthes. DBPL Language and System Manual. ESPRIT FIDE technical report, March 1991.

[Ston87] M. Stonebraker. The Design of the POSTGRES Storage System. Proc. 13th VLDB Conf., Brighton, England, pp.289-300, 1987.

[SRH90] M. Stonebraker, L.A. Rowe, M. Hirohama. The Implementation of POSTGRES. Electronics Research Laboratory, College of Engineering, University of California, Berkeley, Memorandum No. UCB/ERL M90/34, 1990

[Ston91] M. Stonebraker. Managing Persistent Objects in a Multi-level Store. Proc. of SIGMOD 91 Conf., 1991, pp.2-11

[SuMi86] K. Subieta, M. Missala. Semantics of query languages for the Entity-Relationship Model. Proc. 5th Conf. on Entity-Relationship Approach, Dijon, France, pp.197-216, 1986.

[SuMi87] K. Subieta, M. Missala. Data manipulation in NETUL. Proc. 6th Conf. on Entity-Relationship Approach, New York,pp. 391-407, 1987.

[Subi91] K. Subieta. LOQIS: The Object-Oriented Database Programming System Proc.1st Intl. East/West Database Workshop on Next Generation Information System Technology, Kiew, USSR 1990 Springer Lecture Notes in Computer Science, Vol.504, pp.403-421, 1991.

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

[Subi90] K. Subieta Object-Based Virtual Memory for PC-s, The Programmer Manual. Institute of Computer Science Polish Academy of Sciences Report 696, 1990.

[That90] S. Thatte. Persistent Memory: Merging AI-knowledge and Databases. Readings in Object-Oriented Data Base Systems (S.B. Zdonik, D. Maier, Eds.). Morgan-Kaufman, pp.242-250, 1990

[Ullm88] J.D. Ullman. Principles of Database and Knowledge-Base Systems, Vol.I. Computer Science Press 1988

[VKC86] P. Valduriez, S. Khoshafian, G. Copeland. Implementation Technique of Complex Objects. Proc. 12th VLDB Conf., Kyoto, Japan, pp. 101-110, 1986.

[VBD89] F. Valez, G. Bernard, V. Darnis. The O_2 Object Manager: an Overview. Proc. 15th VLDB Conf., Amsterdam, Holland, pp. 357-366, 1987.

[WoKi87] D. Woelk, W. Kim. Multimedia Information Management in an Object-Oriented Database System. Proc. 13th VLDB Conf., Brighton, England, pp.319-330, 1987.

[ZdMa90] S.B. Zdonik, D. Maier. Fundamentals of Object-Oriented Databases. Readings in Object-Oriented Database Systems (S.B. Zdonik, D. Maier, Eds.). Morgan Kaufman, pp.1-32, 1990